(2)

ADVANCED ADA WORKSHOP

Sponsored By
Ada Software Engineering
Education and Training
(ASEET)
Team

Keesler Air Force Base
24-27 January 1989

AGENDA FOR THE Ada SOFTWARE ENGINEERING EDUCATION AND TRAINING (ASEET) TEAM ADVANCED Ada WORKSHOP KEESLER AFB, BILOXI, MS JANUARY 24-27, 1989

TUESDAY - 24 JANUARY

8:30-9:00	Bldg 1002 Room 111	Welcoming Remarks General Announcements Capt Roger Beauman
9:00-12:00 (Break at 10:15)		Software Engineering Capt David Vega Capt Michael Simpson Keesler AFB
12:00-1:30		Lunch
1:30-4:30 (Break at 2:30)		Software Engineering
6:30-8:00	Keesler AFB Officers' Club	Reception

WEDNESDAY - 25 JANUARY

9:00-12:00 (Break at 10:15)	Bldg 1002 Room 111	Generics LCDR Lindy Moran US Naval Academy Major Chuck Engle SEI
12:00-1:30		Lunch
1:30-4:30 (Break at 2:30)		Generics
6:30-11:00 (Optional)	Bldg 1002 Room 148	Hands-on Project Lt Dan O'Donnell Lt Don Princiotta Lt Kevin McGinty Keesler AFB Instructors

THURSDAY - 26 JANUARY

9:00-12:00 (Break at 10:15)	Bldg 1002 Room 111	Tasking Capt David Cook USAF Academy
12:00-1:30		Lunch
1:30-4:30 (Break at 2:30)	Bldg 1002 Room 111	Methodologies for Reuse Capt Eugene Bingue Offutt AFB, Nebraska

FRIDAY - 27 JANUARY

9:00-12:00 (Break at 10:15)	Bldg 1002 Room 111	Exceptions Major Pat Lawlis AFIT

12:00 - End of Workshop

INTRODUCTION TO SOFTWARE ENGINEERING WITH ADA

Captain Michael Simpson

Captain David Vega

Keesler Air Force Base

24 January 1989

OVERVIEW

I. The Software Crisis

II. Program Units

III. Types

IV. Control Statements

V. Exceptions

VI. Generics

VII. Tasks

VIII. Application Example

Software Crisis

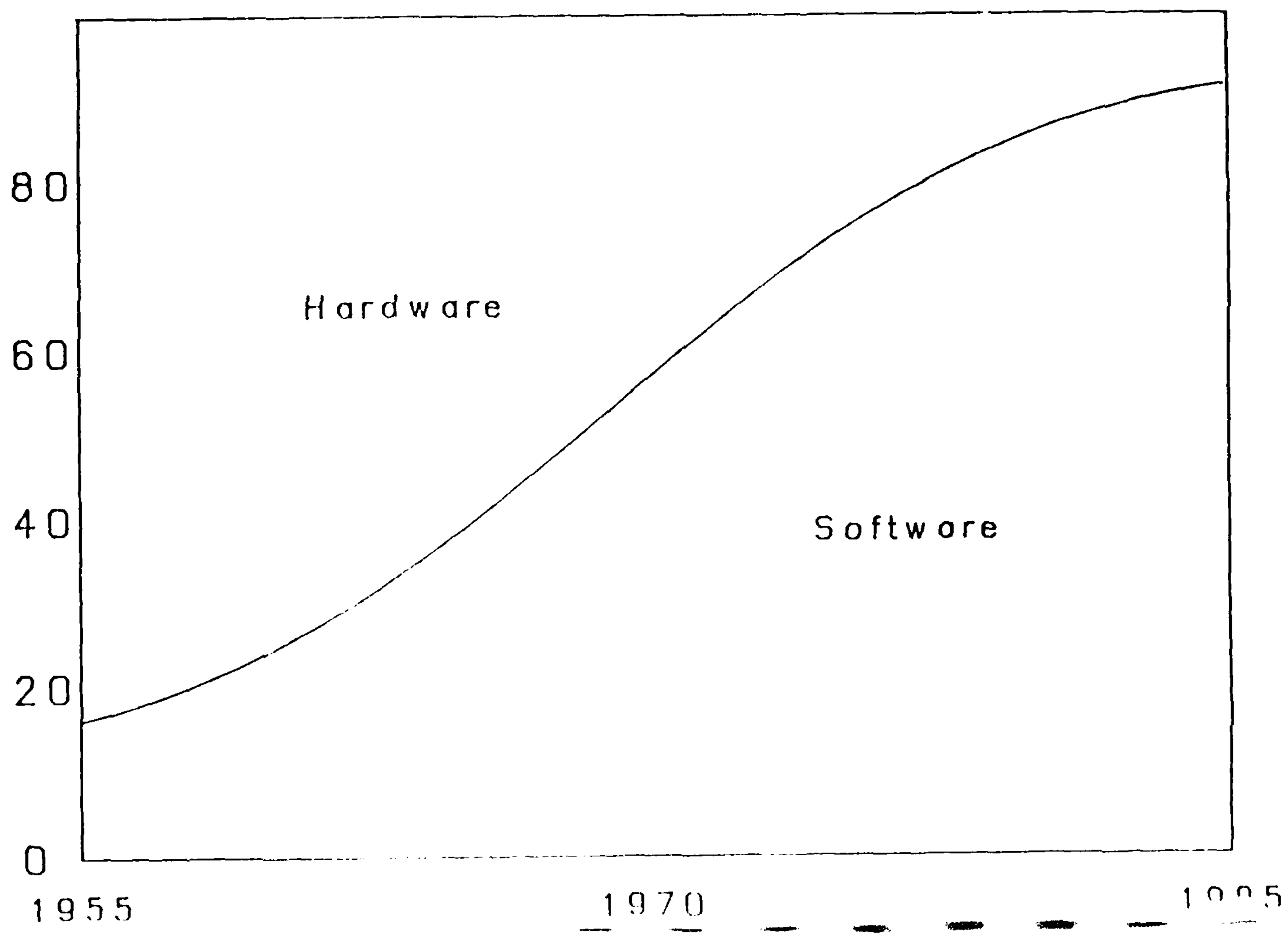
-- Rising costs of software
-- Unreliable
-- Late
-- Not maintainable
-- Inefficient
-- Not transportable

WHY??

-- Too many languages
-- Poor tools
-- Changing technology
-- Not enough trained people

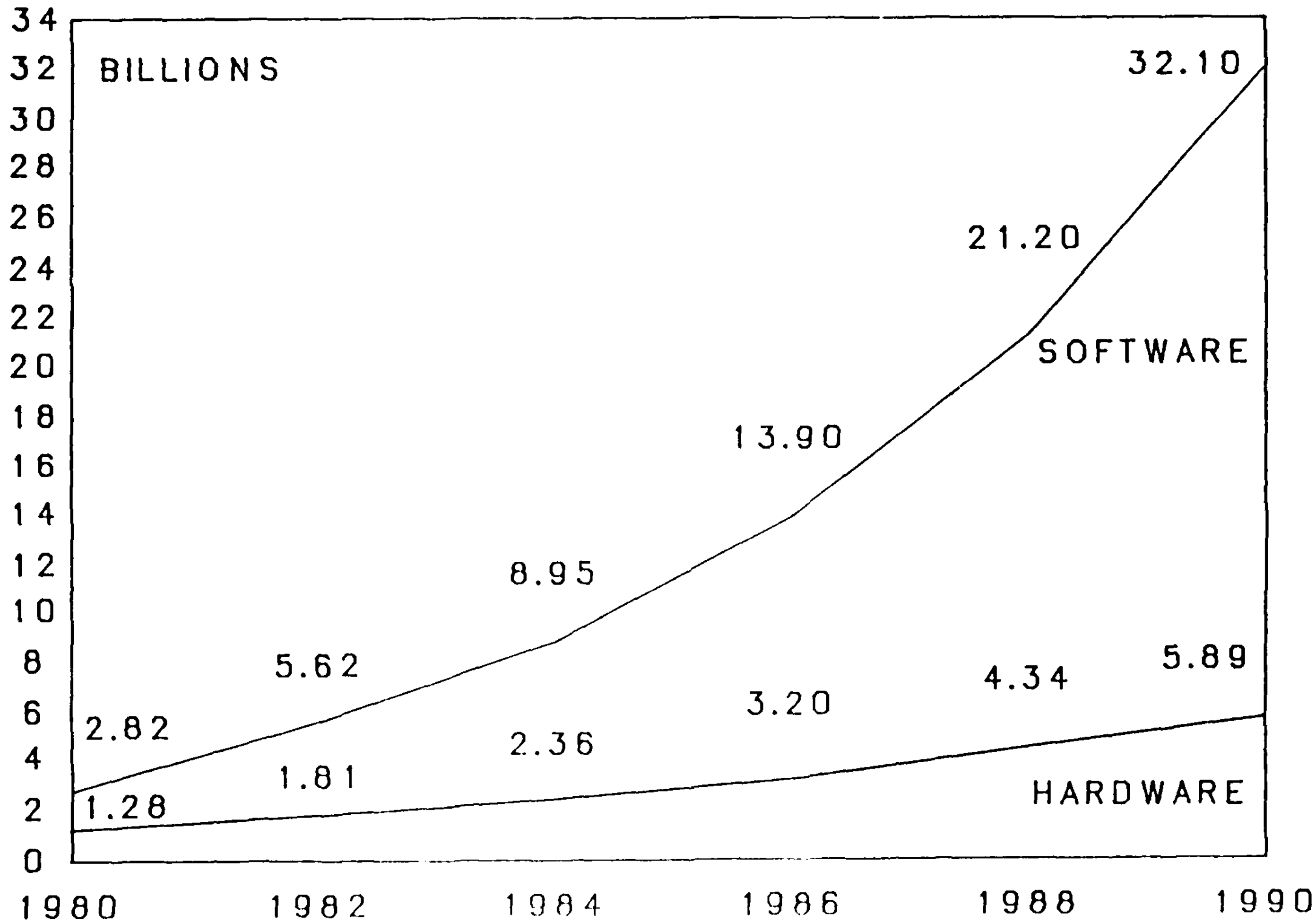
INABILITY TO MANAGE COMPLEX PROBLEMS

Software Crisis

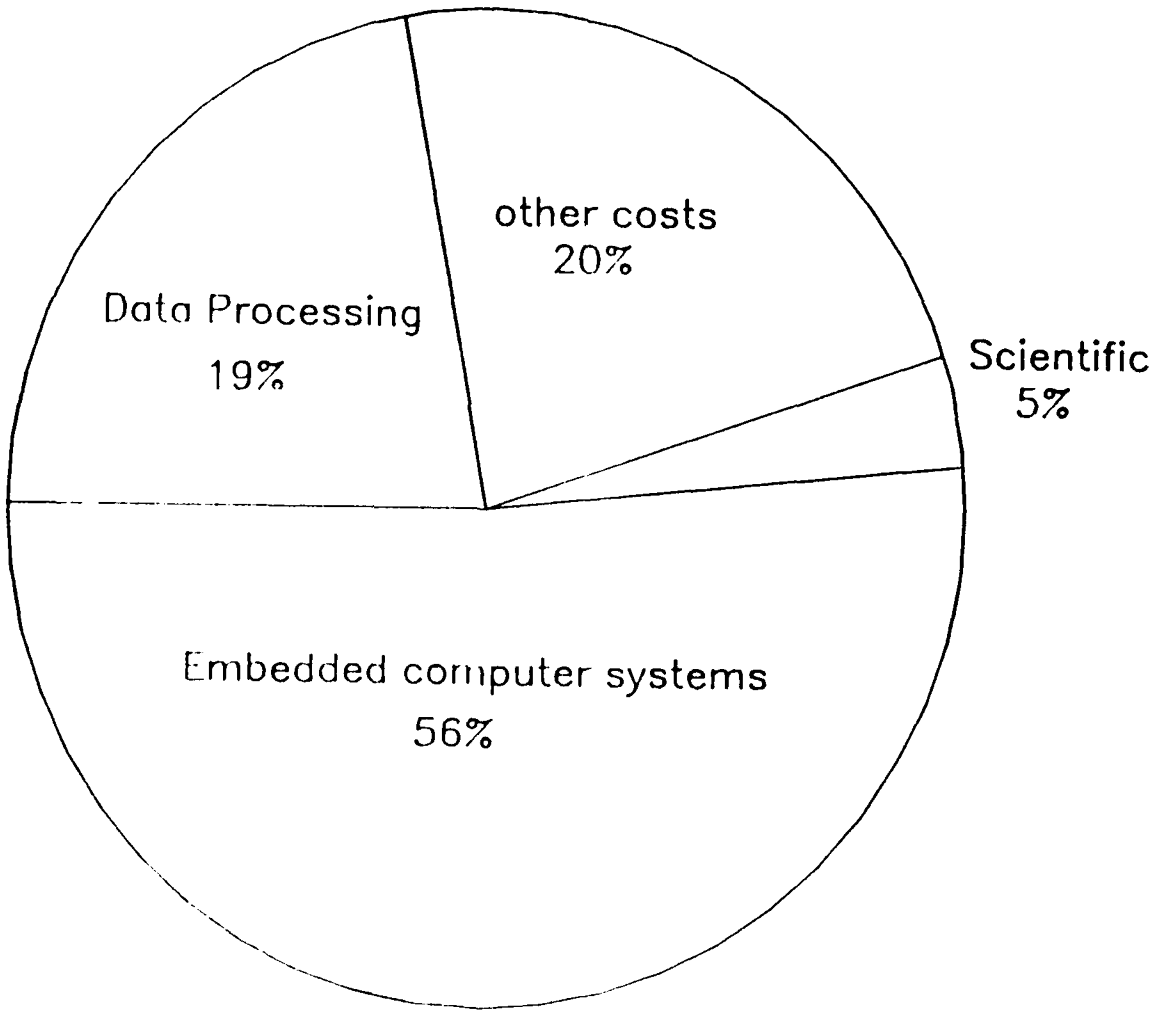


Software Crisis

DoD Embedded Hardware/Software Costs



Software Crisis



Software Crisis

EMBEDDED SYSTEMS

-- Large
-- Long lived
-- Continous change
-- Physical constraints
-- High reliability

EMBEDDED SYSTEMS SOFTWARE

-- Severe reliability requirements
-- Time and size constraints
-- Parallel processing
-- Real time control
-- Exception handling
-- Unique I/O

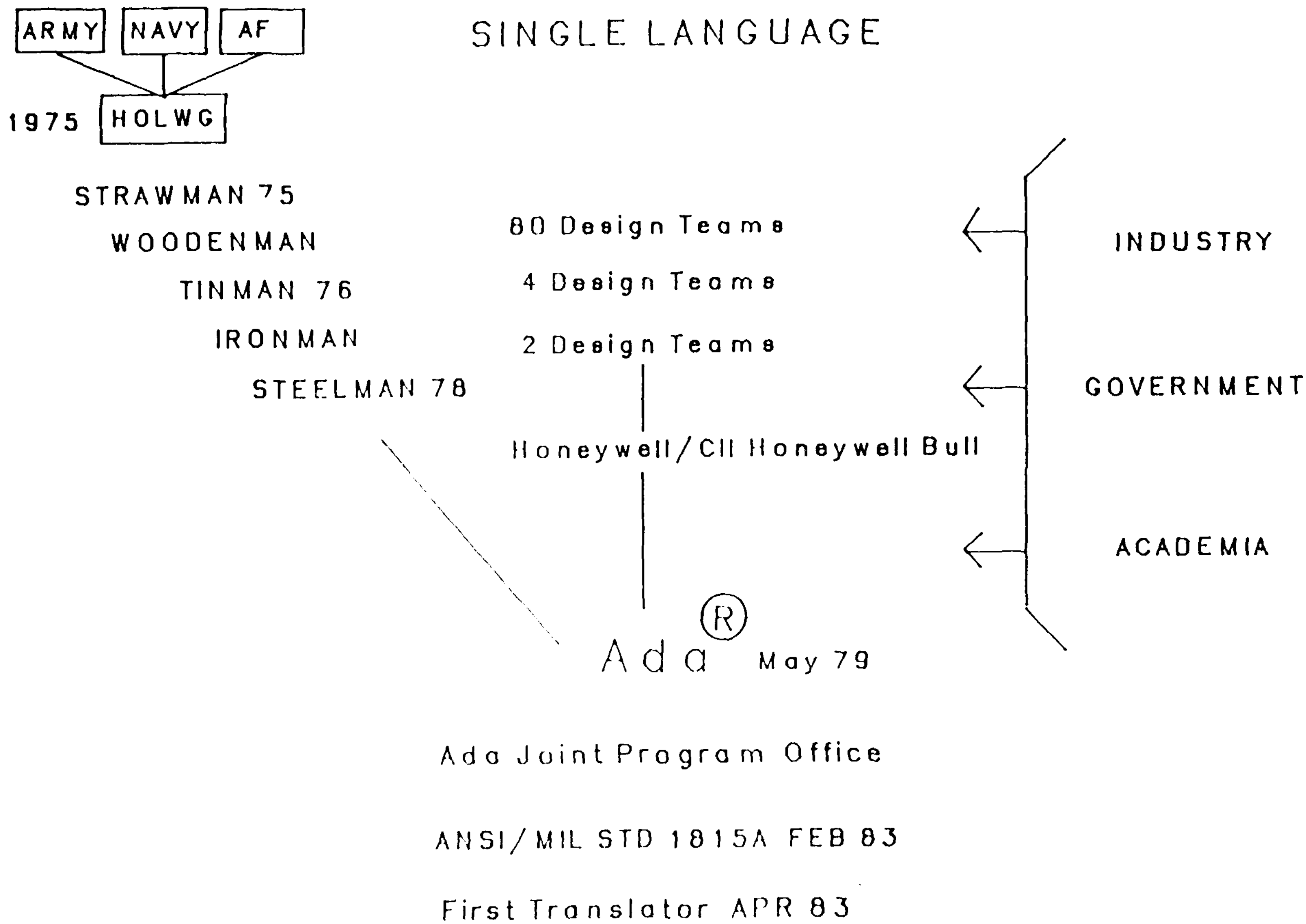
Software Crisis

SOLUTIONS

Single Language	Improved Tools	Improved Methodologies
Ada	Ada	Methodman
	APSE	??
	(Ada Programming Support Environment)	

SOFTWARE ENGINEERING

Software Crisis

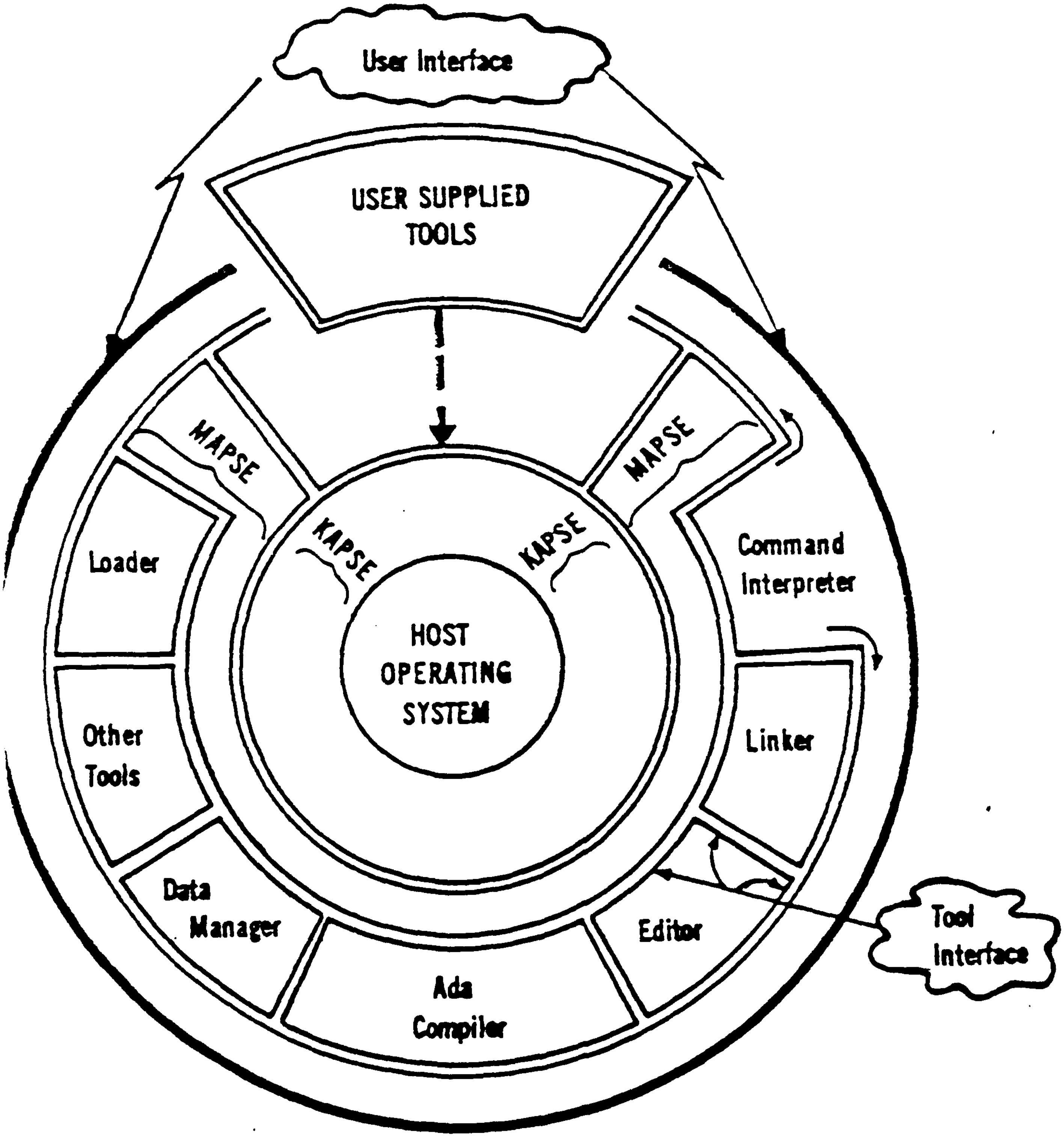


Software Crisis

Ada Programming Support Environment

1978 SANDMAN
PEBBLEMAN
1980 STONEMAN

-- Software developer productivity
-- Retraining costs
-- Lack of tools
-- Lack of standardization


User Interface
USER SUPPLIED
TOOLS
MAPSE
MAPSE
KAPSE
KAPSE
HOST
OPERATING
SYSTEM
Loader
Command
Interpreter
Other
Tools
Linker
Data
Manager
Editor
Ada
Compiler
Tool
Interface

Software Crisis

" The basic problem is not our mismanagement of technology, but rather our inability to manage the complexity of our systems."

-- E.G. Booch

SOFTWARE ENGINEERING

GOALS

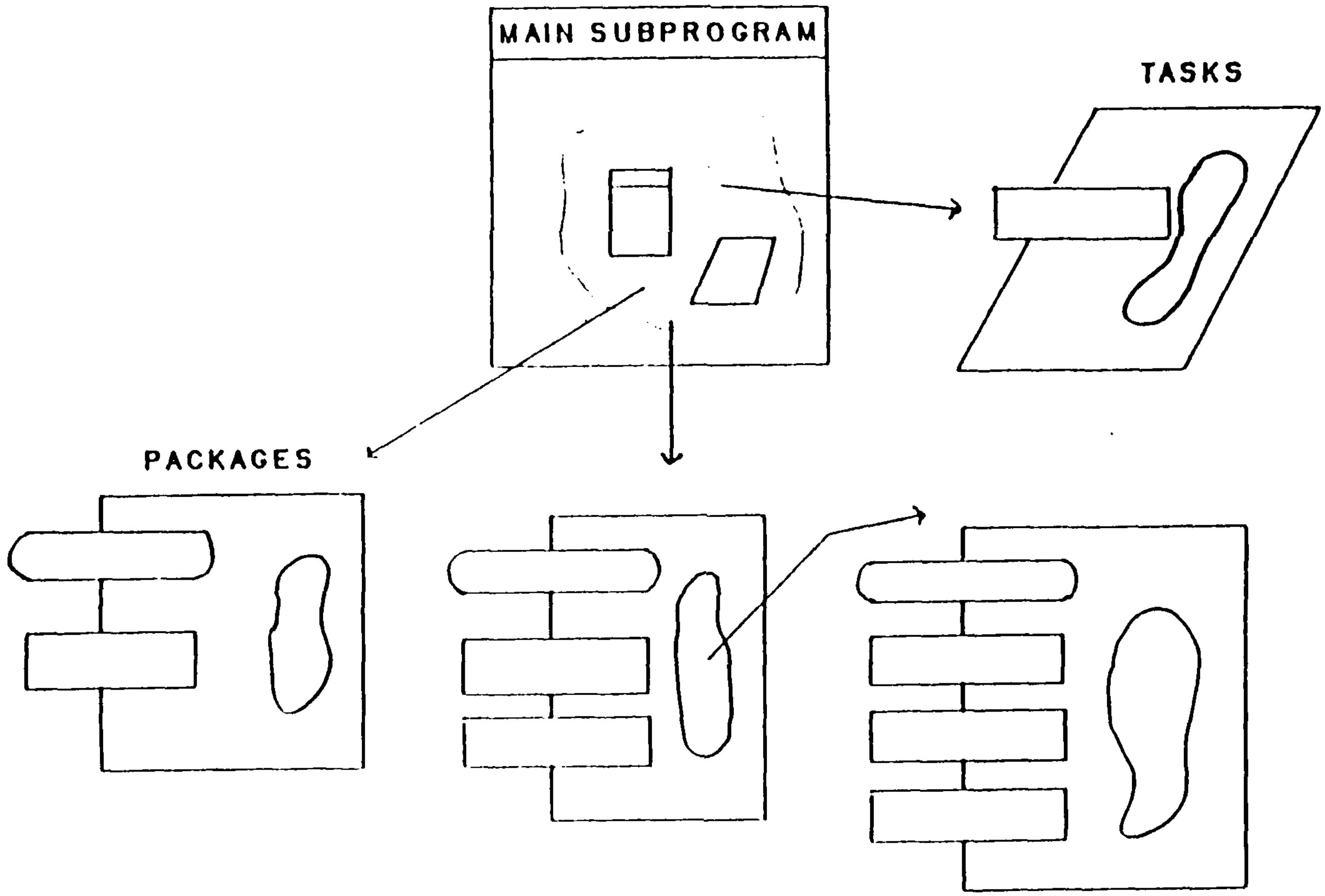
-- Understandabilty
-- Modifiability
-- Reliability
-- Efficiency

PRINCIPLES

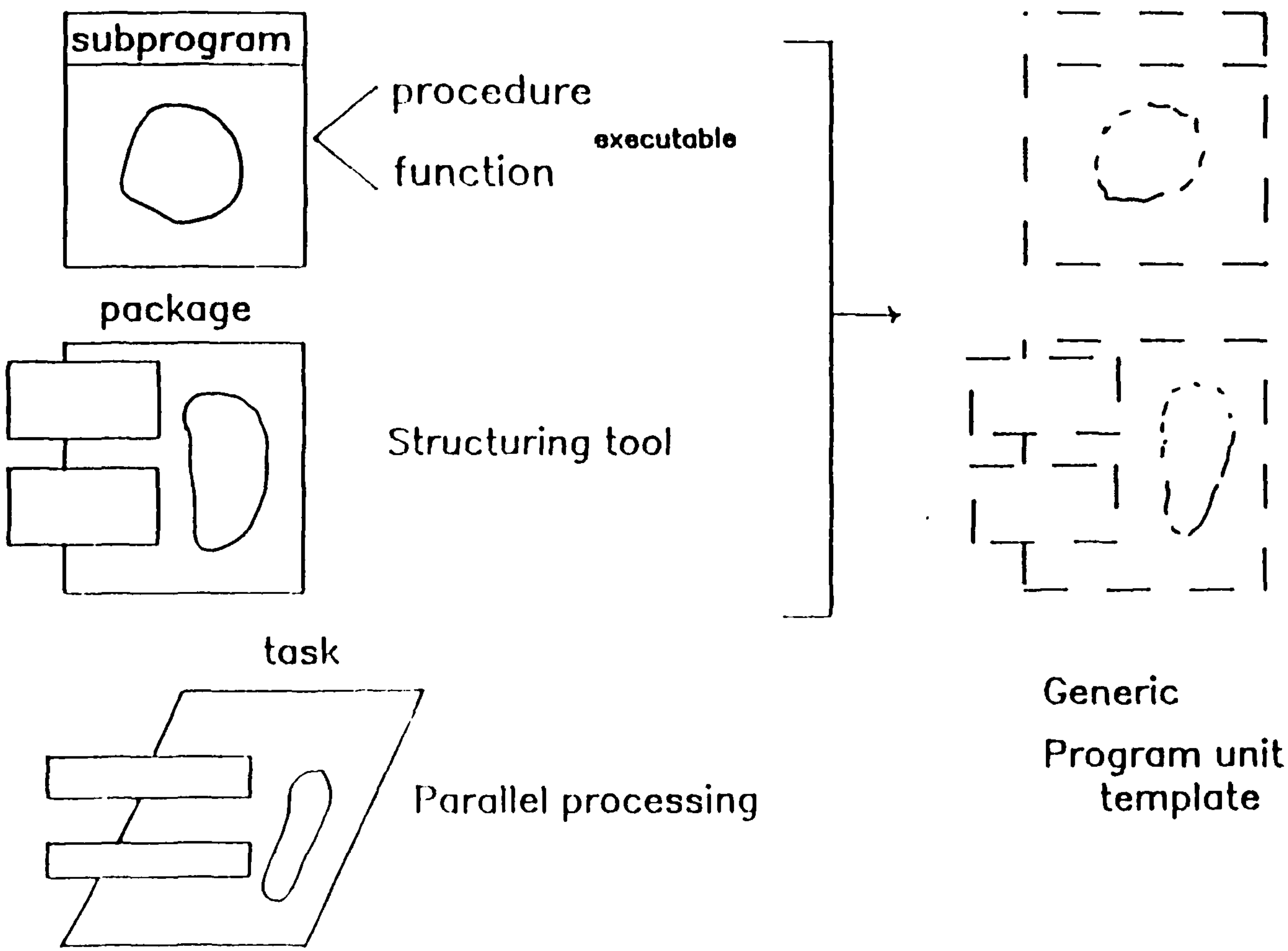
-- Abstraction
-- Information Hiding
-- Modularity
-- Localization
-- Completeness
-- Confirmabilty
-- Consistency

Program Units

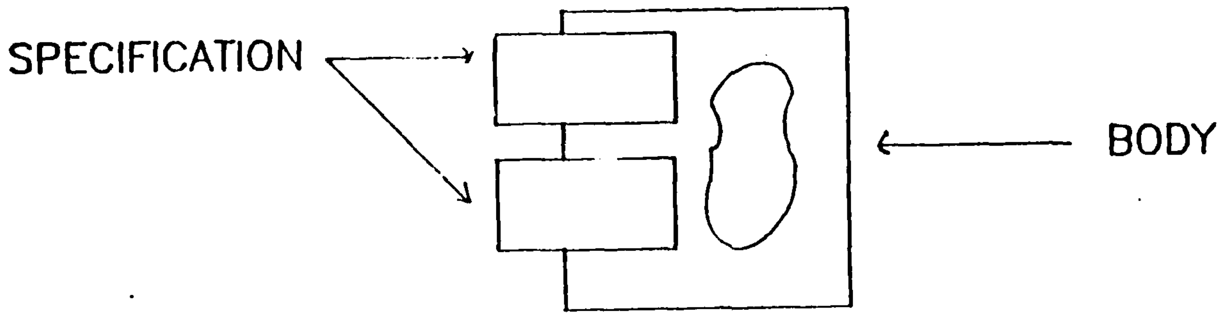
-- Ada software systems consist of one or more program units



Program Units



Program Units



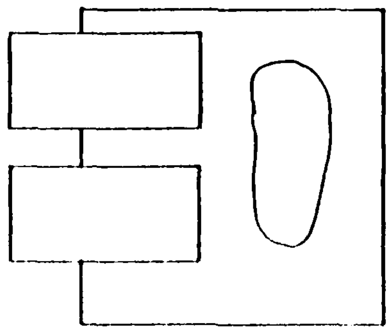
"what" the program unit does $\Longleftarrow$ ABSTRACTION $\Longrightarrow$ "how" the program unit does what it does

all the user of the program unit needs to know $\Longleftarrow$ INFORMATION HIDING $\Longrightarrow$ the details of implementation are inaccesible to the user

Program Units

By separating the "what" from the "how"...

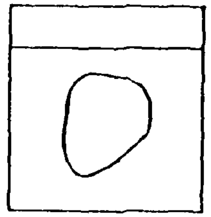
we decrease the complexity of the system...



and increase: UNDERSTANDABILITY
MODIFIABILITY

Program Units

Subprograms



-- Executable routines
-- Main program
-- Recursive

PROCEDURE
-- Defines an action to be performed

```
procedure GET_NAME ( NAME : out STRING );
                    GET_NAME ( PERSONS NAME );
```

FUNCTION
-- Returns a value

```
function SIN ( ANGLE : in RADIANS ) return FLOAT;
            ANGLE_SIN := SIN ( 2 );
```

Program Units

Procedures

SPECIFICATION

-- Defines name

-- Defines parameters to be passed

```
procedure ADD ( FIRST : in INTEGER;
                SECOND : in INTEGER;
                RESULT : out INTEGER );
```

FIRST	:	in	INTEGER
formal parameter name		parameter mode	parameter type

Program Units

Parameter modes

in — The value passed to the subprogram acts as a constant inside and may only be read. Value remains unchanged after completion.

in out — The variable passed to the procedure may be read and updated. Value may change after completion.

out — The variable passed to the procedure may only be updated. Value may change after completion.

Program Units

procedures

BODY

-- Defines the action to be performed
-- Contains a local declarative part
-- Contains a sequence of statements

```
procedure ADD ( FIRST : in INTEGER;
                SECOND : in INTEGER;
                RESULT : out INTEGER ) is
    -- local declarations go here
begin
   RESULT := FIRST + SECOND;
end ADD;
```

Program Units

```
with TEXT_IO;
use TEXT_IO;
procedure MEET_AND_GREET_Ada_IS
   YOUR_NAME : STRING(1..80);
   LAST : NATURAL;

begin
   PUT_LINE("Welcome to the wonderful world of Ada");
   PUT("What is your name? ");
   GET_LINE( YOUR_NAME, LAST );
   PUT("Hi"); PUT ( YOUR_NAME(1..LAST) );
   NEW_LINE;
   PUT_LINE("I hope you like Ada");
end MEET_AND_GREET_Ada;
```

Program Units

```
procedure AN_EXAMPLE is
   MY_INTEGER : INTEGER := 10;
   TEMP       : INTEGER :=  0;

   procedure NEXT (AN_INTEGER : in   INTEGER;
                   VALUE      :out INTEGER) is
   begin
      VALUE := AN_INTEGER + 1;
   end NEXT;
begin
   while MY_INTEGER <= 100 loop
      NEXT(MY_INTEGER,TEMP);
      MY_INTEGER := TEMP;
   end loop;
end AN_EXAMPLE;
```

Program Units

Functions

SPECIFICATION

-- Defines name
-- Defines parameters to be passed
-- Defines result type

```
function ADD ( FIRST, SECOND : in INTEGER )
                            return INTEGER;
```

-- parameter mode can only be "in"
-- called as an expression

Program Units

Functions

BODY

-- Defines the action to be performed

-- Contains a declarative part

-- Contains a sequence of statements

-- Result returned in a "return" statement

```
function ADD ( FIRST, SECOND : INTEGER )
                          return INTEGER is
begin
   return FIRST + SECOND;
end ADD;
```

Program Units

Functions

```
procedure CALCULATIONS is
   VALUE : INTEGER := 1;
   function ADD_PREVIOUS ( NUMBER : in INTEGER )
                              return INTEGER is
   begin
      return NUMBER + ( NUMBER - 1 );
   end ADD PREVIOUS;

begin

   VALUE := ADD_PREVIOUS ( 5 );
   -- value equals 9

end CALCULATIONS;
```

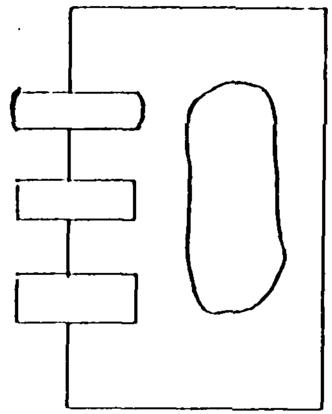
Program Units

Overloading

```
function "*" (LEFT,RIGHT: A_TYPE)
                        return A_TYPE;
  --Overload the "*" operation for A_TYPE

TEMP := "*" (MY_VALUE, YOUR_VALUE);
    -- Prefix notation

TEMP := MY_VALUE * YOUR_VALUE;
    --Infix notation
```

Program Units

Packages



-- Defines groups of logically related items
-- Structuring tool
-- Contains a visible part (specification)
and a hidden part (private part and body)
-- Primary means for extending the language

Program Units

Package specification

```
-- Define items available to user of package ( export )

package CONSTANTS is
   PI : constant := 3.14159;
   e : constant := 2.71828;
   WARP : constant := 3.00E+08;
                        -- meters/second
end CONSTANTS;
```

Program Units

```
package ROBOT_CONTROL is

   type SPEED is range 0..100;
   type DISTANCE is range 0..500;
   type DEGREES is range 0..359;
   procedure GO_FORWARD ( HOW_FAST : in SPEED;
                          HOW_FAR : in DISTANCE );
   procedure REVERSE ( HOW_FAST : in SPEED;
                       HOW_FAR : in DISTANCE );
   procedure TURN ( HOW_MUCH : in DEGREES );

end ROBOT_CONTROL;
```

Program Units

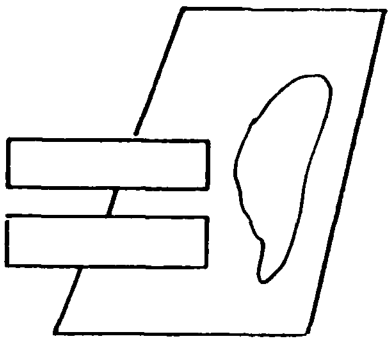
```
with ROBOT_CONTROL; -- Provides access to ROBOT_CONTROL
use  ROBOT_CONTROL;
procedure SQUARE is
begin
   GO_FORWARD ( HOW_FAST => 100, HOW_FAR => 20 );
   TURN ( 90 );
   GO_FORWARD ( 100, 20 );
   TURN ( 90 );
   GO_FORWARD ( 100, 20 );
   TURN ( 90 );
   GO_FORWARD ( 100, 20 );
   TURN ( 90 );
end SQUARE;
```

Program Units

Package bodies

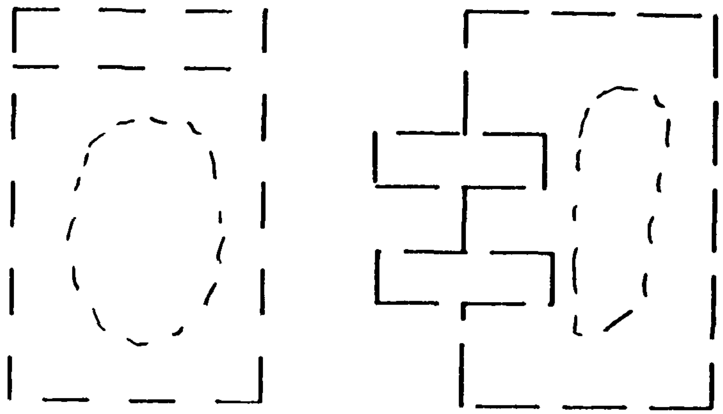
```
--Define local declarations
--Define implementation of subprograms
-- defined in specification
package body ROBOT_CONTROL is
    --local declarations
    procedure RESET_SYSTEM is
    begin
        --implementation
    end RESET_SYSTEM;
    procedure GO_FORWARD...is...
    procedure REVERSE...is...
    procedure TURN...is...
end ROBOT_CONTROL;
```

Program Units



TASK

A program unit that operates in parallel with other program units



GENERIC

Template of a subprogram or package

Types

--A type consists of a set of values that objects of the type may take on, and a set of operations applicable to those values

--Ada is a strongly typed language!

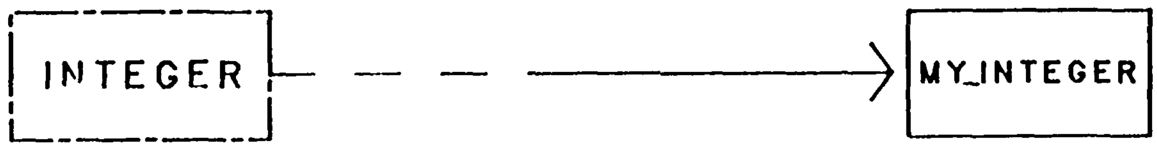
*Every object must be declared of some type name
*Different type names may not be implicitly mixed
*Operations on a type must preserve the type

```
AN_INTEGER      : INTEGER;
A_FLOAT_NUMBER  : FLOAT ;
ANOTHER_FLOAT   : FLOAT;

A_FLOAT_NUMBER := ANOTHER_FLOAT + AN_INTEGER;
                  --illegal
```

Types

Types and Objects

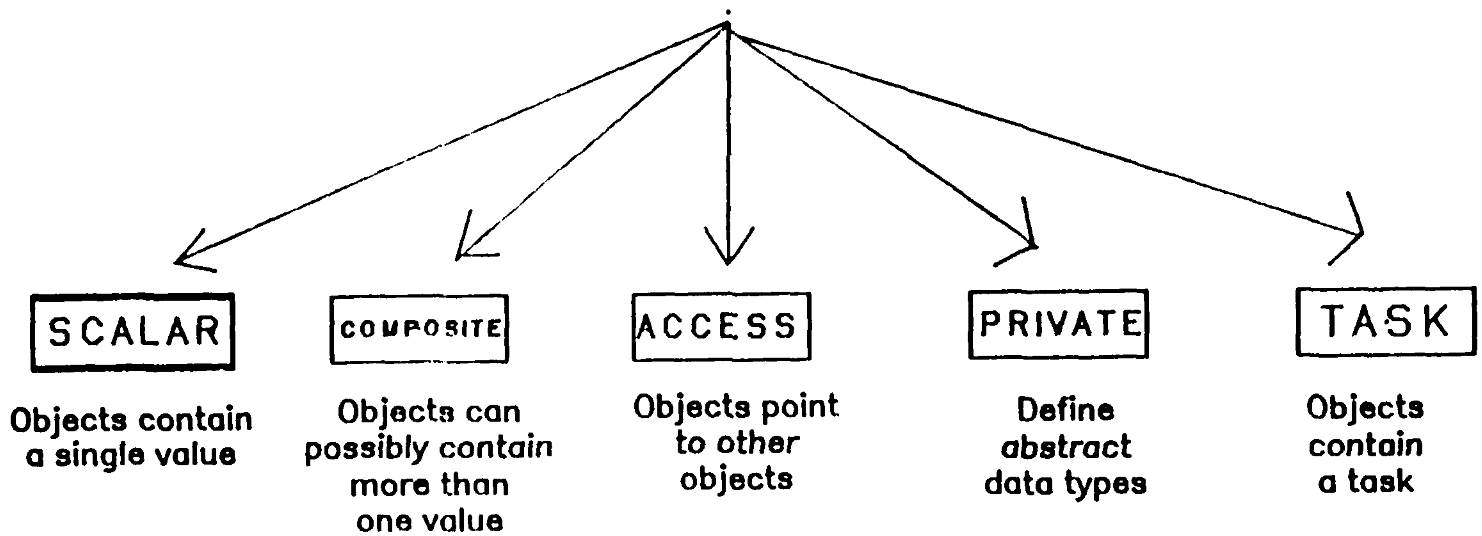
TYPES	OBJECTS
Define a template for objects	Variables or constants that are instances of a type



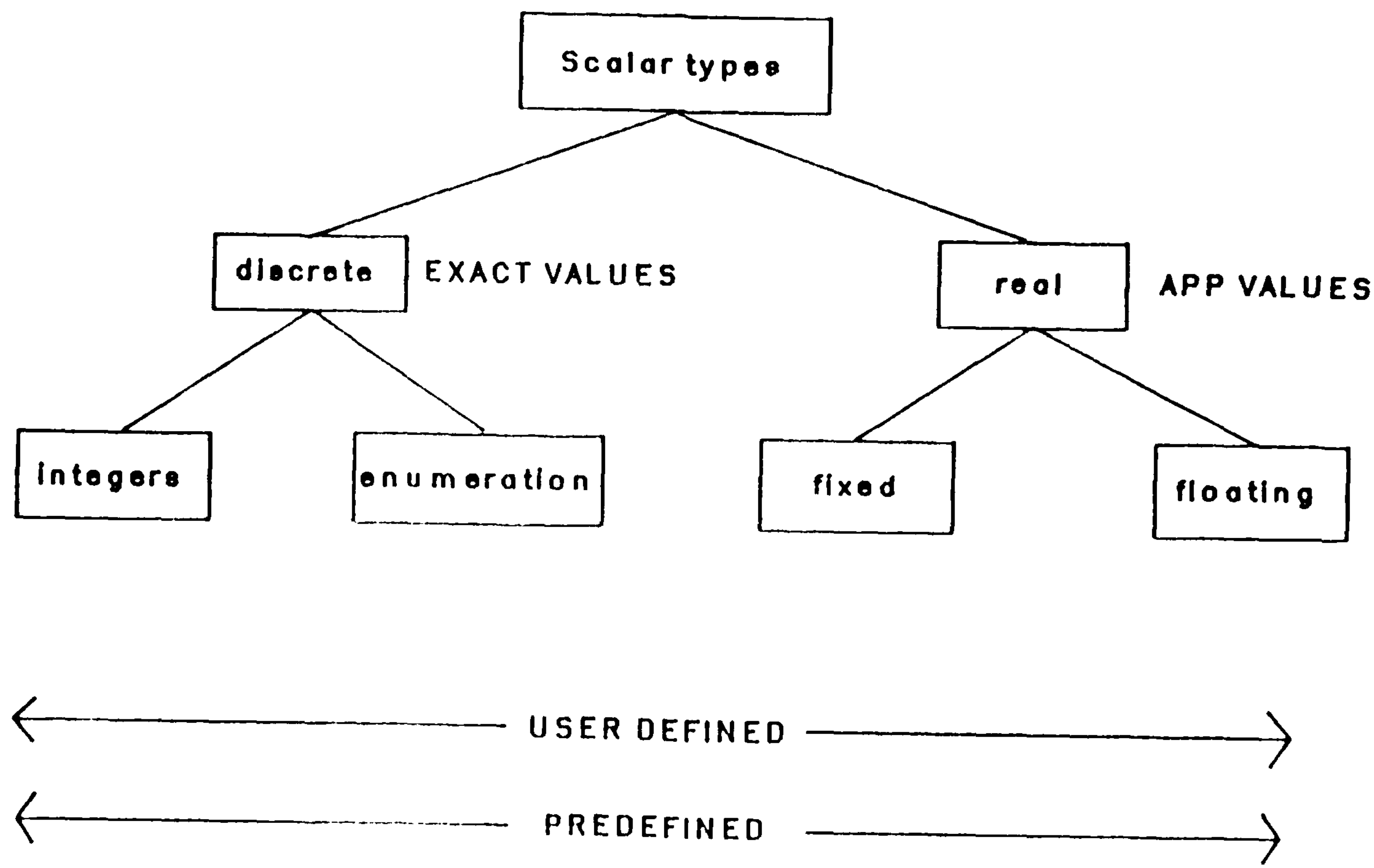
OBJECT DECLARATION

```
MY_INTEGER : INTEGER;
YOUR_INTEGER : INTEGER := 10;
```

Ada Types


SCALAR
Objects contain a single value
COMPOSITE
Objects can possibly contain more than one value
ACCESS
Objects point to other objects
PRIVATE
Define *abstract* data types
TASK
Objects contain a task

Types



Scalar types
discrete
EXACT VALUES
real
APP VALUES
integers
enumeration
fixed
floating
USER DEFINED
PREDEFINED

Types

Integers

--Define a set of exact, consecutive values

USER DEFINED

```
        type ALTITUDE is range 0..100_000;
        type DEPTH is range 0..20_000;
        PLANES_HEIGHT : ALTITUDE;
        DIVER_DEPTH : DEPTH;

 begin
      PLANES_HEIGHT := 10_000;
      PLANES_HEIGHT := 200_000;  -- error
      PLANES_HEIGHT := DIVER_DEPTH; -- error
 end;
```

Types

Predefined integer types

INTEGER------------------->(usually −32,768..32767)

"subtypes" of INTEGER

NATURAL(0..INTEGER'LAST)

POSITIVE(1..INTEGER'LAST)

LONG_INTEGER------------------->(usually double word)

SHORT_INTEGER------------------->(usually half word)

Types

Subtypes

-- Constrain a range of values or accuracy on a type
-- Does not define a new type ,i.e., compatible
with base type

```
type ALTITUDE is range 0..200_000;
subtype HIGH is ALTITUDE range 40_000 .. 200_000;
subtype MEDIUM is ALTITUDE range 10_000 .. 100_000;
subtype LOW is ALTITUDE range 0 .. 10_000;
```

Types

Enumeration

--Define a set of ordered enumeration values
--Used in array indexing, case statements,
-- and looping

USER DEFINED

```
type SUIT is (CLUBS, HEARTS, DIAMONDS, SPADES);
type COLOR is (RED, WHITE, BLUE);
type SWITCH is (OFF, ON);
type EVEN DIGITS is ('2','4','6','8');
type MIXED is (ONE,'2',THREE,'*','!',more);
```

where CLUBS < HEARTS < DIAMONDS < SPADES
(pos 0) (pos 1) (pos 2) (pos 3)

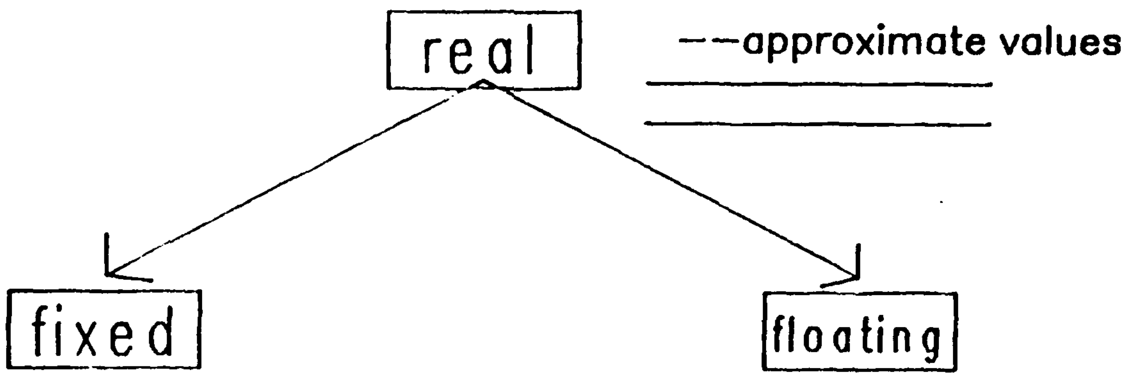
Types

Pre—defined enumeration types

BOOLEAN ---------------------> (FALSE, TRUE)

CHARACTER

Types



fixed point arithmetic

floating point arithmetic

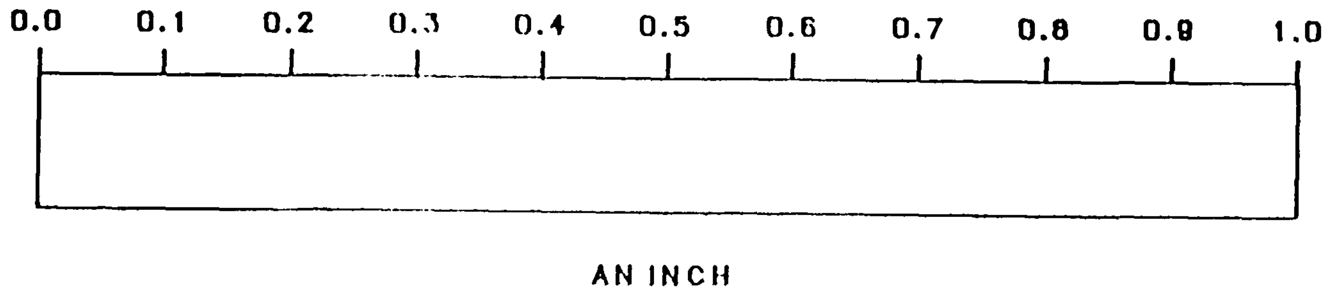
Types

Fixed point types

-- Absolute bound on error
-- Larger error for smaller numbers (around zero)

USER DEFINED

type TENTHS_OF_INCH is delta 0.1 range 0.0 .. 1.0;



PREDEFINED

DURATION --> (Used for "delay" statements)

Types

Floating point types

-- Relative bound of error
-- Defined in terms of significant digits
-- More accurate at smaller numbers, less at larger

USER DEFINED

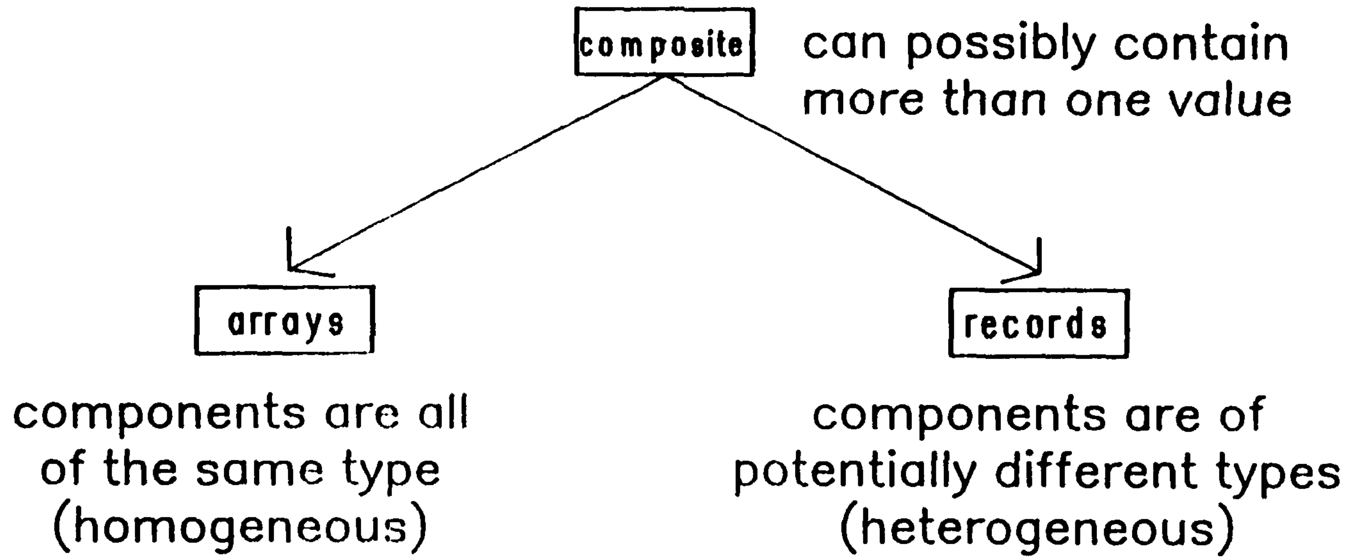
```
type NUMBERS is digits 3 range 0.0 .. 20_000;
```

0.001, 0.002, 0.003...999.0,1000.0,1010.0...,10000.0,10100.0

PREDEFINED

FLOAT

Types


composite
can possibly contain more than one value
arrays
components are all of the same type (homogeneous)
records
components are of potentially different types (heterogeneous)

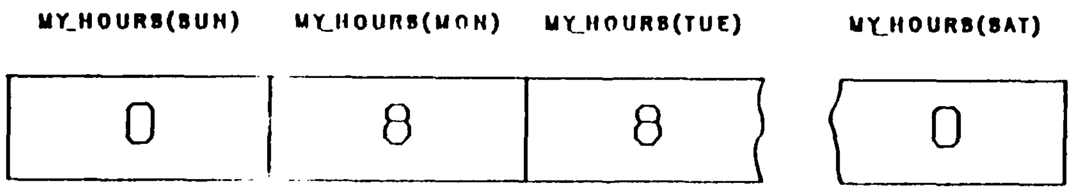
Types

Arrays: constrained, unconstrained

CONSTRAINED

-- Indices are static for all objects of that type

```
type HOURS is range 0..40;
type DAYS is ( SUN,MON,TUE,WED,THU,FRI,SAT );
type WORK_HOURS is array( DAYS ) of HOURS;

MY_HOURS : WORK_HOURS := ( 0,8,8,7,6,1,0 );
```



Types

Arrays

UNCONSTRAINED

--Indices are known at elaboration (run) time

--Indices may be different for different objects

```
type HOURS is range 0..40;
type DAYS is (SUN,MON,TUE,WED,THU,FRI,SAT);
type WORK_HOURS is array (DAYS range <>) of HOURS;

HOLIDAY_WEEK : WORK_HOURS (TUE..SAT) :=(others =>0);
FULL_WEEK : WORK_HOURS (DAYS'FIRST..DAYS'LAST);
```

Types

```
procedure DAYS_WORKED (FIRST,SECOND: in DAYS) is

    A_WEEK : WORK_HOURS (FIRST..SECOND);

  begin
    .
    .
    .
  DAYS_WORKED(WED,FRI);
```

	WED	THU	FRI
A_WEEK			

```
  DAYS_WORKED(FRI,SAT);
```

	FRI	SAT
A_WEEK		

Types

Multi-dimensional arrays

```
    type VALUES is digits 6 range -10.0 .. 100.0;
    type INDEX is range 1..3;
    type TWO_D_MATRIX is array (INDEX,INDEX) of VALUES;

    MY_MATRIX : TWO_D_MATRIX := ( others =>(others => 0.0));
    IDENTITY_MATRIX : constant TWO_D_MATRIX := ( (1.0,0.0,0.0),
                                                 (0.0,1.0,0.0),
                                                 (0.0,0.0,1.0));

begin

    MY_MATRIX := IDENTITY_MATRIX;
    MY_MATRIX (3,3) := 2.0;
    .
    .
    .
```

Types

Array

PREDEFINED

```
type STRING is array (POSITIVE range <>) of CHARACTER;
```

USE OF THE PREDEFINED STRING TYPE

```
YOUR_STRING : STRING (1..10);
MY_STRING : STRING (1..20);
THEIR_STRING : STRING; -- illegal
```

STRING SLICING

```
YOUR_STRING := MY_STRING(1..10);
MY_STRING(11..15) := YOUR_STRING(2..6);
MY_STRING(3..4) := MY_STRING(4..5);
MY_STRING(2) := 'G';
MY_STRING(2) := "G";  -- illegal
```

Types

Records

undiscriminated
discriminated
variant

```
UNDISCRIMINATED
   type DAYS is ( MON,TUE,WED,THU,FRI,SAT,SUN );
   type DAY is range 1..31;
   type MONTH is (JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,
                  SEP,OCT,NOV,DEC);
   type YEAR is range 0..2085;
   type DATE is record
      DAY_OF_WEEK : DAYS;
      DAY_NUMBER : DAY;
      MONTH_NAME : MONTH;
      YEAR_NUMBER : YEAR;
   end record;
   TODAY : DATE;
begin
   TODAY.DAY_OF_WEEK := TUE;
   TODAY.DAY_NUMBER := 26;
   TODAY.MONTH_NAME := NOV;
```

TODAY

DAY_OF_WEEK	TUE
DAY_NUMBER	26
MONTH_NAME	NOV
YEAR_NUMBER	1985

Types

Records

```
    type A_MONTH is array (DAY range <>) of DATE;
    NOVEMBER: A_MONTH(1..30);

begin

    NOVEMBER(26).DAY_OF_WEEK := TUE;
    NOVEMBER(27) := (WED,27,NOV,1985);
```

Types

Records

DISCRIMINATED

```
    type BUFFER(SIZE:POSITIVE := 10) is record
       ITEMS : STRING(1..SIZE);
    end record;

    MY_BUFFER : BUFFER; -- size is 10;
    YOUR_BUFFER : BUFFER (20);
    THEIR_BUFFER : BUFFER (SIZE => 15);

begin
    MY_BUFFER.ITEMS := "Hi There!!";
```

Types

Records

VARIANT

```
type DRIVER is (GOOD,BAD);
type INSURANCE_RATE is range 1..50;
type DISCOUNT is delta 0.01 range 0.0..1.0;
type INSURANCE (KIND:DRIVER) is record
   NORMAL_RATE : INSURANCE_RATE;
   case KIND is
        when GOOD => DISCOUNT_RATE : DISCOUNT ;
        when BAD => ADDITIONAL : INSURANCE_RATE;
    end case;
 end record;
```

Types

Records

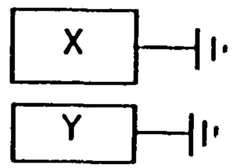
```
    A_DRIVER : INSURANCE (GOOD);
    ANOTHER : INSURANCE(BAD);

begin
  A_DRIVER.NORMAL_RATE := 25;
  A_DRIVER.DISCOUNT_RATE := 0.15;

  ANOTHER.NORMAL_RATE := 25;
  ANOTHER.ADDITIONAL := 10;
```

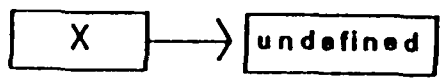
Types

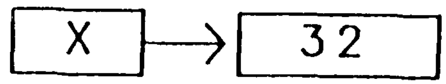
Access

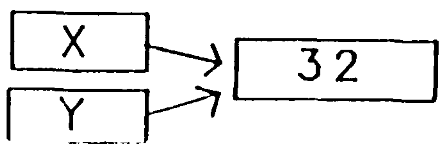
-- Pointer variables
-- Allow for dynamic allocation of memory
-- Objects created via an allocator

```
type POINTER is access INTEGER;

X, Y : POINTER;   -- initialized to
                  -- null
begin

X := new INTEGER; -- allocate
                  -- memory to X
X.all := 32; -- place 32 in the
             -- location pointed to
             -- by X
Y := X; -- X and Y point to the same
        -- location
```





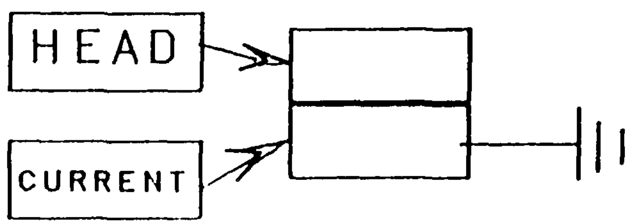




Types

Access types – Linked list

```
procedure LINKED LIST is
  type ITEM;  – incomplete type declaration
  type POINTER is access ITEM;
  type ITEM is record
        NAME: STRING(1..20):=(others =>' ');
        NEXT : POINTER;
   end record;

   HEAD,CURRENT,TEMP:POINTER; --initialized to null

begin

   HEAD:=new ITEM;
   CURRENT:=HEAD;
```



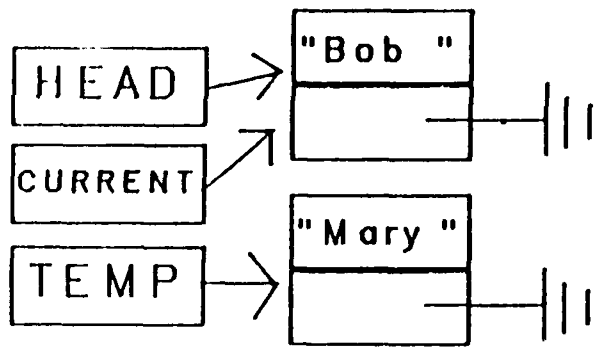

```
   CURRENT.NAME(1..3):= "Bob";
```

Types

Access types – Linked list

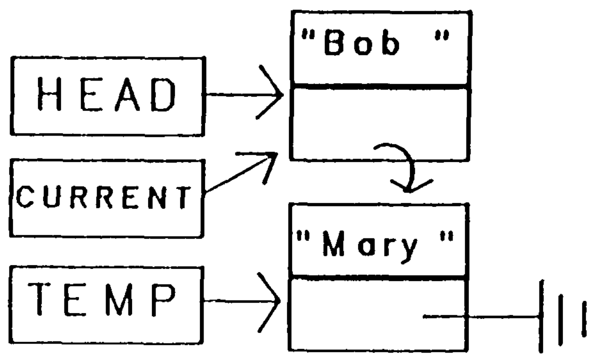
Create a New Item

```
TEMP := new ITEM;
TEMP.NAME(1..4):="MARY ";
```



Add to List

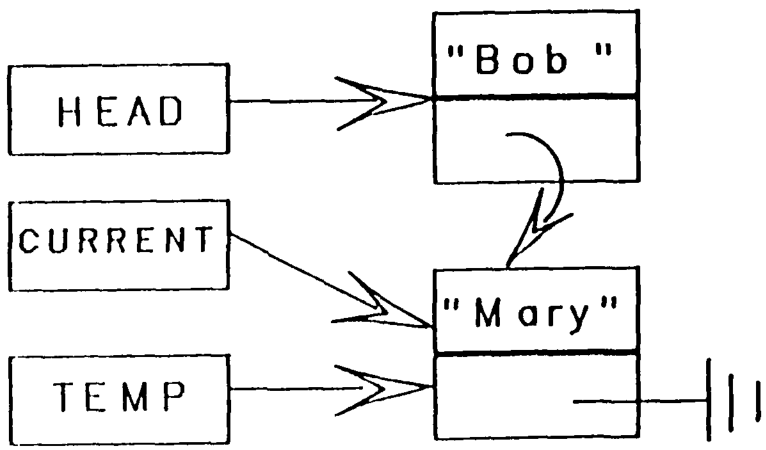
```
CURRENT.NEXT:=TEMP;
```



Types

Access types— Linked list

```
--Move current pointer
CURRENT := TEMP;
```

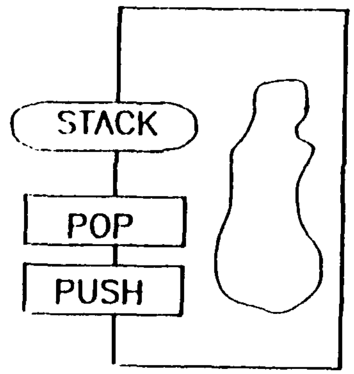


Types

Private types

-- Defined in a package
-- Used to create abstract data types
-- Used to extend the language
-- Directly supports abstraction and
-- Information hiding

INTEGER_STACK



PRIVATE

:= = /=

subprograms defined in
package specification

LIMITED PRIVATE

only subprograms
defined in
package specification

Types

```
package B_R is
   type NUMBERS is range 0..99;
   procedure TAKE ( A_NUMBER : out NUMBERS );
   function NOW_SERVING return NUMBERS;
   procedure SERVE ( NUMBER : in NUMBERS );
end B_R;
package body B_R is
   SERV_A_MATIC : NUMBERS := 1;
   procedure TAKE ( A_NUMBER : out NUMBERS ) is
   begin
      A_NUMBER := SERV_A_MATIC;
      SERV_A_MATIC := SERV_A_MATIC + 1;
   end TAKE;
   function NOW_SERVING return NUMBERS is separate;
   procedure SERVE ( NUMBER : in NUMBERS ) is
                                            separate;

end B_R;
```

Types

```
with B_R; use B_R;
procedure ICE_CREAM is
   YOUR_NUMBER : NUMBERS;
begin
   TAKE ( YOUR_NUMBER );
   loop
      if NOW_SERVING = YOUR_NUMBER then
         SERVE ( YOUR_NUMBER );
         exit;
      end if;
   end loop;
end ICE_CREAM;
```

```
PUT_LINE("What is the name of the file to output to?");
GET_LINE( FILE_NAME, LAST );

CREATE ( SORTED_FILE, OUT_FILE, FILE_NAME(1..LAST) );

for FILE_ITEM in 1 .. LIST_INDEX - 1 loop

  PUT_LINE( SORTED_FILE,THE_LIST(FILE_ITEM).NAME );
  PUT LINE(SORTED FILE,THE LIST(FILE ITEM).ADDRESS );
  PUT_LINE(SORTED_FILE,THE_LIST(FILE_ITEM).PHONE_NUMBER);
  NEW_LINE(SORTED_FILE);

 end loop;

 CLOSE ( SORTED_FILE );

end ORDER_LIST;
```

```
begin

 PUT_LINE ("This program sorts a list of names, addresses and ");
 PUT_LINE ("phone numbers and puts that sorted list in a file.");
 NEW_LINE (2);
 PUT_LINE ("What is the name of the file to sort?");
 GET_LINE (FILE_NAME, LAST);
 OPEN (UNSORTED_FILE, IN_FILE, FILE_NAME (1..LAST));

 while not END_OF_FILE (UNSORTED_FILE) loop

  GET_LINE (UNSORTED_FILE, THE_LIST (LIST_INDEX).NAME, LAST);
  GET_LINE (UNSORTED_FILE, THE_LIST (LIST_INDEX).ADDRESS, LAST);
  GET_LINE (UNSORTED_FILE, THE_LIST (LIST_INDEX).PHONE_NUMBER, LAST);
  LIST_INDEX := LIST_INDEX + 1;

 end loop;

 SORT (THE_LIST (1..LIST_INDEX - 1));
 CLOSE (UNSORTED_FILE);
```

```
with LIST_PACKAGE, TEXT_IO;
use  LIST_PACKAGE, TEXT_IO;
procedure ORDER_LIST is

   UNSORTED_FILE : FILE_TYPE;
   SORTED_FILE : FILE_TYPE;

   MAX_ITEMS : constant := 20;

   THE_LIST : A_LIST(1..MAX_ITEMS);
   LIST_INDEX : POSITIVE := 1;

   LAST : NATURAL;
   FILE_NAME : STRING(1..40);
```

```
with SWAP;
package body LIST_PACKAGE is

  procedure SWAP_ITEMS is new SWAP ( ELEMENT_TYPE => ITEMS );

  procedure SORT ( ANY_LIST : in out A_LIST ) is

    -- implements a selection sort
    SMALLEST_INDEX, TEMP_INDEX : POSITIVE;
    SMALLEST_NAME : A_LINE := ( others => ' ' );

  begin
    for SORTED_INDEX in ANY_LIST'RANGE loop
      SMALLEST_INDEX := SORTED_INDEX;
      for CHECK_INDEX in (SORTED_INDEX+1)..ANY_LIST'LAST loop
        if ANY_LIST ( CHECK_INDEX).NAME <
                  ANY_LIST (SMALLEST_INDEX).NAME then
          SMALLEST_INDEX := CHECK_INDEX;
          SWAP_ITEMS ( ANY_LIST(SMALLEST_INDEX),
                       ANY_LIST(SORTED_INDEX) );
        end if;
      end loop;
    end loop;
  end SORT;

end LIST PACKAGE;
```

```
package LIST_PACKAGE is

  MAX_LINE_LENGTH : constant := 80;

  subtype A_LINE is STRING(1..MAX_LINE_LENGTH);

  type ITEMS is record
    NAME : A_LINE := ( others => ' ' );
    ADDRESS : A_LINE := ( others => ' ' );
    PHONE_NUMBER := ( others => ' ' );
  end record;

  type A_LIST is array( POSITIVE range <> ) of ITEMS;

  procedure SORT ( ANY_LIST : in out A_LIST );

end LIST_PACKAGE;
```

Tasks

SELECT

```
--Used to choose between entries in a task
  task DRIVE_CONTROL is
       entry READ(DATA: out DATA_TYPE);
       entry WRITE(DATA: in DATA_TYPE);
  end DRIVE_CONTROL;

  task body DRIVE_CONTROL is
  begin
       loop
            select
                 accept READ(DATA:out DATA_TYPE)do
                 .
                 end;
            or
                 accept WRITE(DATA:in DATA_TYPE)do
                 .
                 end;
            end select;
       end loop;
  end DRIVE_CONTROL;
```

Tasks

DELAY

--Used to suspend execution for at least
--   the time interval specified

```
delay 30.0;
```

ABORT

--Used to unconditionally terminate a task
--Only used in extreme circumstances

```
abort CHANNEL;
```

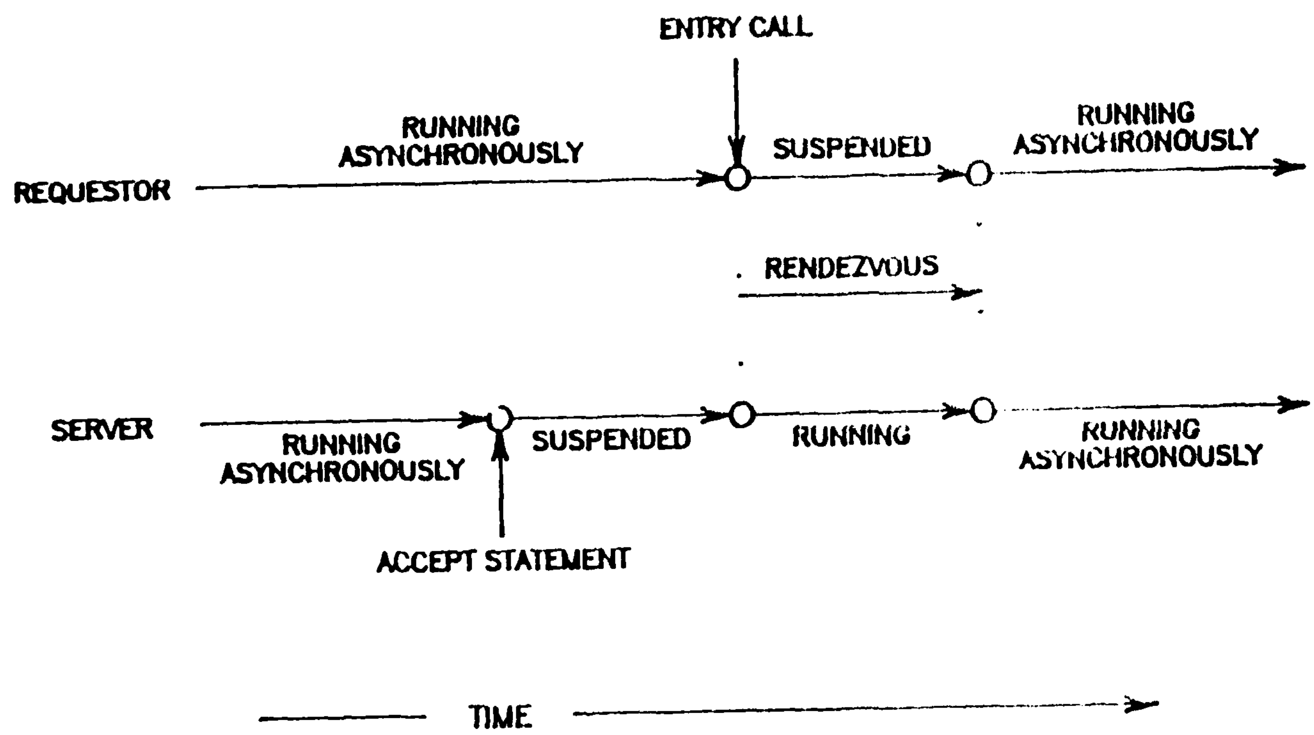
Tasks

Tasking statements

ENTRY CALL
DELAY
ABORT
ACCEPT
SELECT

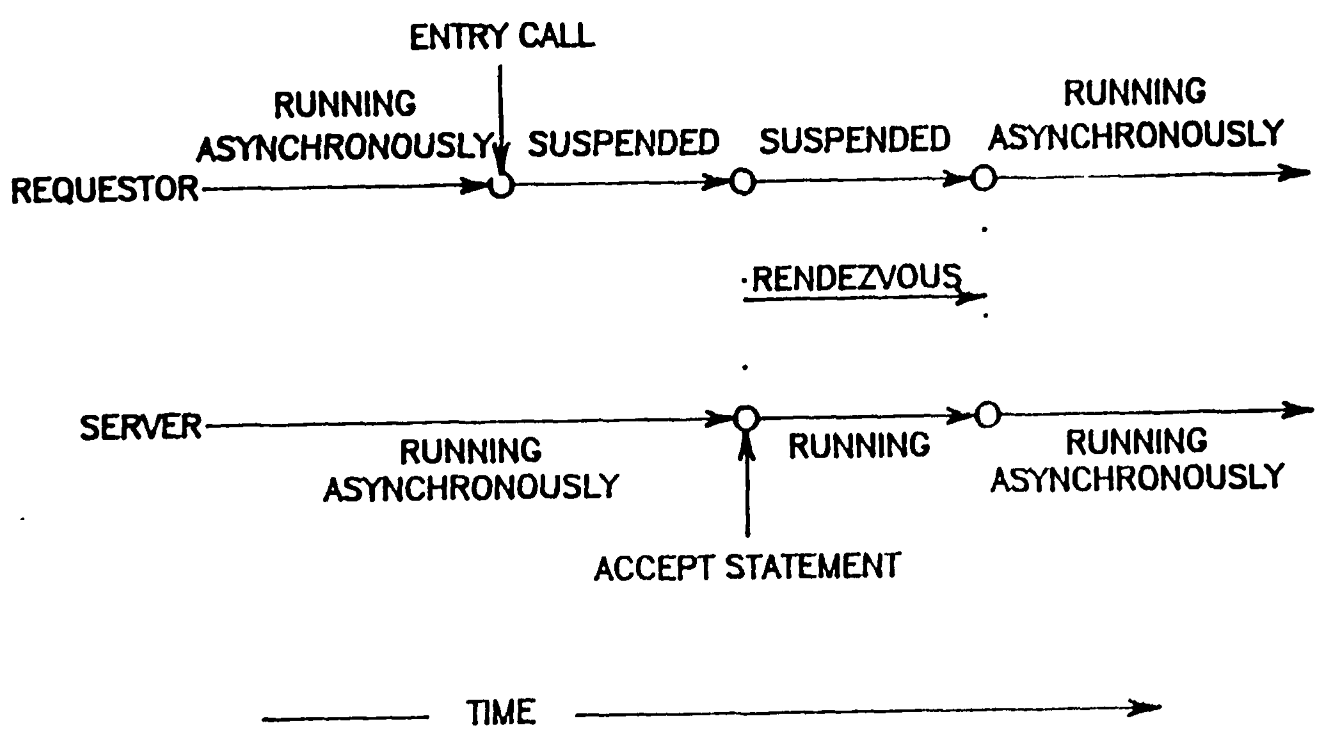
Tasks

STAGES OF A RENDEZVOUS (ACCEPT FIRST)



Tasks

STAGES OF A RENDEZVOUS (ENTRY CALL FIRST)



Tasks

```
--Inside a task, rendezvous occurs when
--    a task's entry has been called and
--    an accept statement is reached

   task body CHANNEL is
         LOCAL_NUMBER : JOB_NUMBER;
   begin
         loop
               accept PRINT(JOB:in JOB_NUMBER)do
                     LOCAL_NUMBER := JOB;
               end;
               CALL_PRINTER (LOCAL_NUMBER);
         end loop;
   end CHANNEL;
```

Tasks

```
--Tasks can communicate with each other
--    via parameters defined in entries

        task CHANNEL is
              entry PRINT(JOB:in JOB_NUMBER);
         end CHANNEL;

--To communicate use an "entry" call

          CHANNEL.PRINT(24);


--When two tasks are synchronized in time
--    and are communicating, we say that the
--    two tasks are in "rendezvous"
```

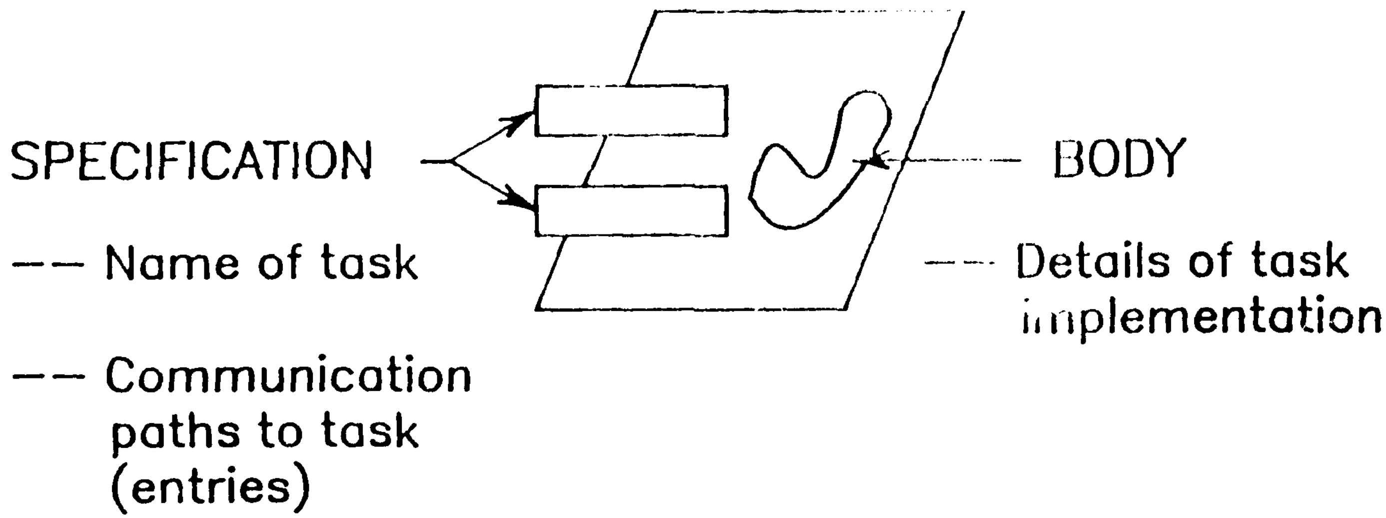
Tasks

```
-- a basic task with no communication
 with TEXT_IO; use TEXT_IO;
 procedure COUNT_NUMBERS is
       package INT_IO is new INTEGER_IO (INTEGER);
       use INT_IO;
       task COUNT_SMALL;
       task COUNT_LARGE;

       task body COUNT_SMALL is
       begin
            for INDEX in -100..0 loop
                 PUT(INDEX);
                 NEW_LINE;
             end loop;
        end COUNT_SMALL;

        task body COUNT_LARGE is
        begin
             for INDEX in 0..100 loop
                  PUT(INDEX);
                  NEW_LINE;
               end loop;
           end COUNT_LARGE;

    begin
       null;  --tasks are started here
```

Tasks

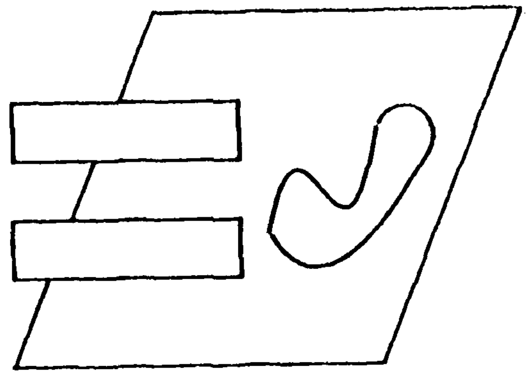
```
procedure SENSOR_CONTROLLER is

    function OUT_OF_LIMITS return BOOLEAN;
    procedure SOUND_ALARM;


    task MONITOR_SENSOR; -- specification
    task body MONITOR_SENSOR is  -- body
    begin
          loop
              if OUT_OF_LIMITS then
                 SOUND_ALARM;
              end if;
          end loop;
    end MONITOR_SENSOR;

    function OUT_OF_LIMITS return BOOLEAN is separate;
    procedure SOUND_ALARM is separate;
begin
   null;  -- Task is activated here
end SENSOR_CONTROLLER;
```

Tasks



SPECIFICATION
-- Name of task
-- Communication paths to task (entries)
BODY
--- Details of task implementation

Tasks



- A task is an entity that operates in parallel with other entities
- Tasking may be implemented on
  - Single Processors
  - Multi-processors
  - Multi-computers

Generics

```
with SQUARING;
procedure MATH_PROGRAM_2 is

   type MATRIX is array (1..3, 1..3) of INTEGER;
   A_MATRIX : MATRIX :=
                 (others => (others => 2));
   function MULT (LEFT, RIGHT : MATRIX) return
                         MATRIX is separate;

   function SQUARE_A_MATRIX is new SQUARING
                        (MATRIX, MULT);

begin
   A_MATRIX := SQUARE_A_MATRIX (A_MATRIX);
end MATH_PROGRAM_2;
```

Generics

```
with SQUARING;
procedure MATH_PROGRAM is

   function SQUARE is new SQUARING (INTEGER);
   X : INTEGER := 8;

begin

   X := SQUARE (X);

end MATH_PROGRAM;
```

Generics

```
generic

    type ELEM is private;
    with function "*" (LEFT, RIGHT : ELEM)
                              return ELEM is < >;

function SQUARING (X : ELEM) return ELEM;
function SQUARING (X : ELEM) return ELEM is

begin

    return X * X;

end SQUARING;
```

Generics

```
with STACK, TEXT_IO; use TEXT_IO;
procedure STACK_OPS_2 is

   STACK_SIZE : POSITIVE := 50;
   INT_ELEMENT : POSITIVE;
   FLOAT_ELEMENT : FLOAT;
   package INT_IO is new INTEGER_IO (POSITIVE);
   package REAL_IO is new FLOAT_IO (FLOAT);
   package INT_STACK is new STACK (STACK_SIZE, POSITIVE);
   package FLOAT_STACK is new STACK (100, FLOAT);
   use INT_IO, REAL_IO, INT_STACK, FLOAT_STACK;
begin
   PUT ("Enter a positive element to push on the stack: ");
   GET (INT_ELEMENT);
   PUSH (INT_ELEMENT);
   PUT ("Enter a FLOAT element to push on the stack: ");
   GET (FLOAT_ELEMENT);
   PUSH (FLOAT_ELEMENT);

end STACK_OPS_2;
```

Generics

```
with STACK;
with TEXT_IO; use TEXT_IO;
procedure STACK_OPS is
   package INT_IO is new INTEGER_IO (POSITIVE);
   use INT_IO;
   INT_ELEMENT : POSITIVE;
   STACK_SIZE : POSITIVE := 50;
   package INTEGER_STACK is new STACK
                                (STACK_SIZE, POSITIVE);
   use INTEGER_STACK;
begin
   PUT ("Enter an element to push on the stack: ");
   GET (INT_ELEMENT);
   PUSH (INT_ELEMENT);
   POP (INT_ELEMENT);
   PUT ("The element popped off the stack was: ");
   PUT (INT_ELEMENT);
```

Generics

```
package body STACK is
     SPACE: array (1..SIZE) of ELEMENT;
     TOP:INTEGER range 0..SIZE:= 0;
     procedure PUSH(ITEM:in ELEMENT)is
     begin
         if TOP = SIZE then
                 raise STACK_OVERFLOW;
         end if;
         TOP := TOP +1;
         SPACE(TOP) := ITEM;
    end PUSH;

    procedure POP(ITEM:in out ELEMENT) is
    begin
        if TOP = 0 then
                raise STACK_UNDERFLOW;
        end if;
        ITEM := SPACE(TOP);
        TOP := TOP -1;
        end POP;
  end STACK;
```

Generics

```
generic
      SIZE: in POSITIVE;
      type ELEMENT is private;

package STACK is

      STACK_UNDER_FLOW,
      STACK_OVER_FLOW : exception;
      procedure PUSH (ITEM:in ELEMENT);
      procedure POP (ITEM:in out ELEMENT);

end STACK;
```

Generics

```
with NEXT;
with TEXT_IO; use TEXT_IO;
procedure MAIN_DRIVER_2 is
   type HOUR is range 1..12;
   THIS_HOUR, NEXT_HOUR : HOUR;
   package HOUR_IO is new      INTEGER_IO (HOUR);
   function HOUR_AFTER is new NEXT (HOUR);

 begin

   PUT ("The current hour is: ");
   HOUR_IO.GET (THIS_HOUR);
   NEXT_HOUR := HOUR_AFTER(THIS_HOUR);
   PUT ("Next hour is: ");
   HOUR_IO.PUT (NEXT_HOUR);

 end MAIN_DRIVER_2;
```

Generics

```
with NEXT;
with TEXT_IO; use TEXT_IO;
procedure MAIN_DRIVER is
   type DAYS is (MON, TUE, WED, THUR, FRI, SAT, SUN);
   TODAY, TOMORROW : DAYS;
   package DAYS_IO is new ENUMERATION_IO (DAYS);
   function DAY_AFTER is new NEXT (DAYS);

 begin

   PUT ("Enter the day: ");
   DAYS_IO.GET (TODAY);
   TOMORROW := DAY_AFTER (TODAY);
   PUT ("Tomorrow is: ");
   DAYS_IO.PUT (TOMORROW);

 end MAIN_DRIVER;
```

Generics

```
generic
   type DISCRETE_TYPE is (<>);
function NEXT(VALUE : in DISCRETE_TYPE)
                 return DISCRETE_TYPE;
function NEXT(VALUE : in DISCRETE_TYPE)
                 return DISCRETE_TYPE is
begin
  if VALUE = DISCRETE_TYPE'LAST then
     return DISCRETE_TYPE'FIRST
  else
     return DISCRETE_TYPE'SUCC(VALUE);
  end if;
end NEXT;
```

Generics

```
with SWAP;
procedure EXAMPLE is
  procedure INTEGER_SWAP is new SWAP(INTEGER);
  procedure CHARACTER_SWAP is new SWAP(CHARACTER);

  NUM_1, NUM_2 : INTEGER;
  CHAR_1, CHAR_2 : CHARACTER;
begin
  NUM_1 := 10;
  NUM_2 := 25;
  INTEGER_SWAP(NUM_1, NUM_2 );
  CHAR_1 := 'A';
  CHAR_2 := 'S';
  CHARACTER_SWAP(CHAR_1, CHAR_2);
end EXAMPLE;
```

Generics

```
generic
      type ELEMENT is private;
procedure SWAP (ITEM_1,ITEM_2:in out ELEMENT);

procedure SWAP(ITEM_1,ITEM_2:in out ELEMENT) is
      TEMP:ELEMENT;
 begin
      TEMP := ITEM_1;
      ITEM_1 := ITEM_2;
      ITEM_2 := TEMP;
 end SWAP;
```

Generics

```
procedure INTEGER_SWAP (FIRST_INTEGER, SECOND_INTEGER:
                        in out INTEGER) is

        TEMP : INTEGER;

begin

        TEMP           := FIRST_INTEGER;
        FIRST_INTEGER  := SECOND_INTEGER;
        SECOND_INTEGER := TEMP;

end INTEGER_SWAP;
```

Generics

Generics Provide:

- factorization
- reduction in size of program text
- more compact code
- no unnecessary duplication of source
- maintainability
- readability
- efficiency

Generics

Data Objects

- To define the template: use type declaration
- To define an instance: use object declaration

Generic program units

- To define the template: use generic declaration
- To define an instance: use generic instantiation

Generics

Parameterized Program Unit
- subprograms
- packages

Cannot be called

Must be instantiated

Exceptions

USER DEFINED

STACK_OVERFLOW : exception;
BAD_INPUT : exception;
DEAD_SENSOR : exception;

PREDEFINED

CONSTRAINT_ERROR
NUMERIC_ERROR
PROGRAM_ERROR
STORAGE_ERROR
TASKING_ERROR

I/O EXCEPTIONS

STATUS_ERROR
MODE_ERROR
NAME_ERROR
USE_ERROR
DEVICE_ERROR
END_ERROR

Exceptions

```
begin
   loop
      begin
         GET ( A_NUMBER );
         NEW_LINE;
         PUT ( "The number is ");
         PUT ( A_NUMBER );
         NEW_LINE;
      exception
         when DATA_ERROR => PUT_LINE("Bad number, try again");
      end;
   end loop;
end GET_NUMBERS;
```

Exceptions

-- When an exception situation occurs, the exception is said to be "raised"

-- What happens then, depends on the presence or absence of an exception handler

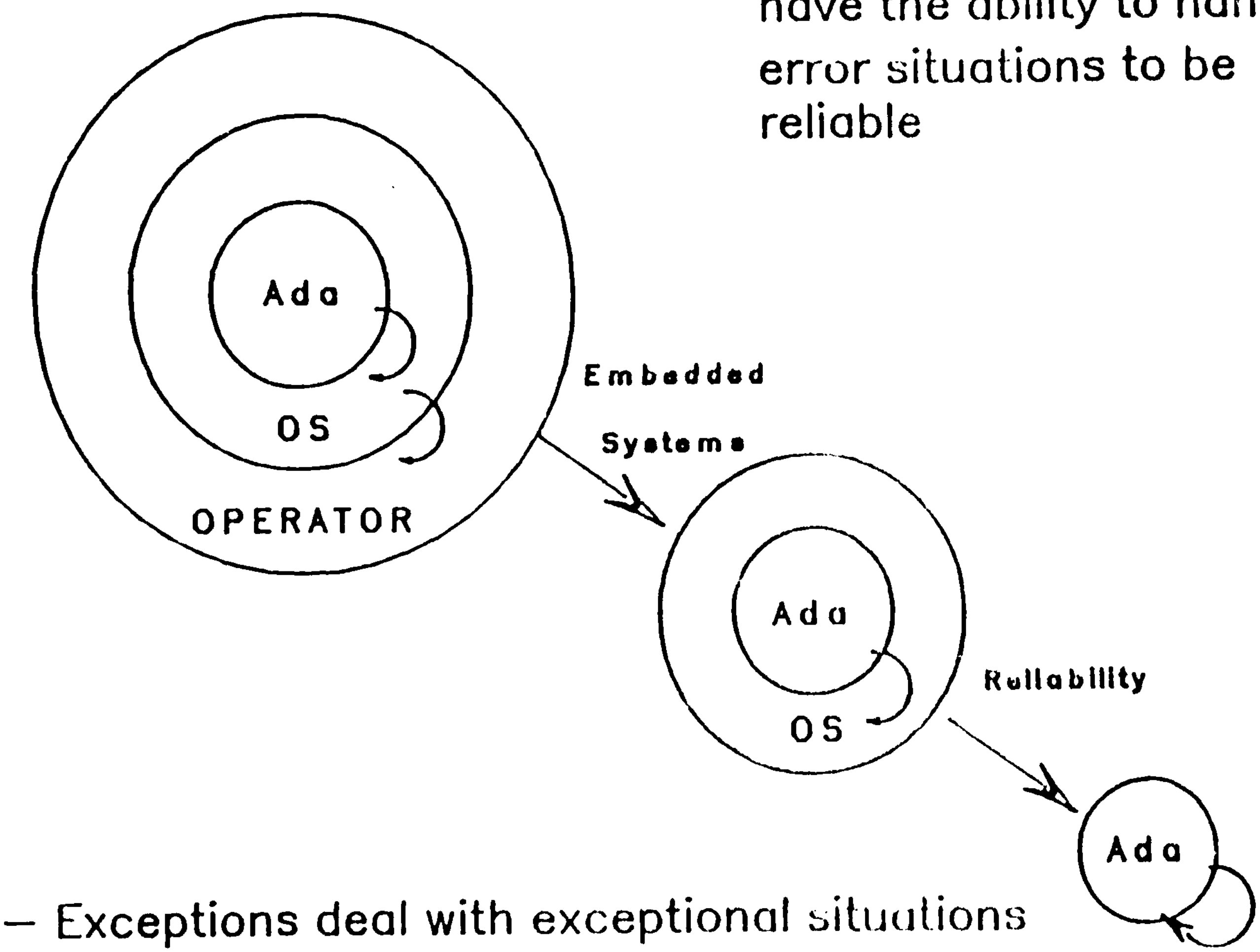
```
begin
   loop
      GET ( A_NUMBER );
      NEW_LINE;
      PUT("The number is");
      PUT ( A_NUMBER );
      NEW_LINE;
   end loop;
end GET_NUMBERS;
```

Exceptions

```
with TEXT_IO; use TEXT_IO;
procedure GET_NUMBERS is
  type NUMBERS is range 1..100;
  package NUM_IO is new INTEGER_IO ( NUMBERS );
  use NUM_IO;
  A_NUMBER : NUMBERS;
begin
  loop
    GET ( A_NUMBER );
    NEW_LINE;
    PUT("The number is ");
    PUT ( A_NUMBER );
    NEW_LINE;
  end loop;
exception
  when DATA_ERROR => PUT_LINE("That was a bad number");
end GET_NUMBERS;
```

Exceptions

-- Real time systems must have the ability to handle error situations to be reliable



-- Exceptions deal with exceptional situations

Control Statements

Iterative

WHILE LOOP ITERATON SCHEME

```
while NOT_DARK loop
   PLAY_TENNIS;
end loop;

TURN_ON_LIGHTS;
```

Control Statements

Iterative

```
for MY_INDEX in 20..40 loop
  -- some statements
end loop;

for YOUR_INDEX in reverse 20..40 loop
  -- some statements
end loop;
```

Control Statements

Iterative

FOR LOOP ITERATION SCHEME

```
with TEXT_IO; use TEXT_IO;
procedure PRINT_ALL_VALUES is
   type COLORS is ( RED, WHITE, BLUE );
   package COLOR_IO is new ENUMERATION_IO ( COLORS );
   use COLOR_IO;

begin

   for INDEX in 1..5 loop
      null;
   end loop;

   for A_COLOR in COLORS loop
     PUT ( A_COLOR );
     NEW_LINE;
   end loop;
end PRINT_ALL_VALUES;
```

Control Statements

Iterative

```
OUTER:
loop

   INNER:
   loop
     if X = 20 then
        exit OUTER;
     end if;
     exit INNER when X = 21;
     X := X + 2;
   end loop INNER;
end loop OUTER;
```

Control Statements

Iterative

BASIC LOOP

```
loop
  -- statements
end loop;
```

EXIT STATEMENT

```
loop
   if X = 20 then
      exit;
   end if;
end loop;
```

```
loop
  loop
    if X = 20 then
      exit;
    end if;
  end loop;
end loop;
```

Control Statements

Conditional

CASE

```
case TIME is
   when EARLY_AM | MID_AM => DRINK_COFFEE;
   when LUNCH => GO_EAT;
   when AFTERNOON => STAY_AWAKE;
   when LATE_AFTERNOON => GET_READY_TO_GO_HOME;
   when others => GET_READY_FOR_TOMMORROW;

end case;
```

Control Statements

Conditional

```
     type DAY_TIMES is ( EARLY_AM,MID_AM,LUNCH,AFTERNOON,
                         LATE_AFTERNOON,DINNER,EVENING,NIGHT );
     TIME : DAY_TIMES := AFTERNOON;
begin
   if TIME = EARLY_AM then
      DRINK_COFFEE;
   elsif TIME = MID_AM then
      DRINK_COFFEE;
   elsif TIME = LUNCH then
      GO_EAT;
   elsif TIME = AFTERNOON then
      STAY_AWAKE;
   elsif TIME = LATE_AFTERNOON then
      GET_READY_TO_GO_HOME;
   else
      GET_READY_FOR_TOMMORROW;
   end if;

end;
```

Control Statements

Conditional

IF

```
if MACHINE_IS_RUNNING then
  SET_NEW_SPEED ( 47 );
elsif MACHINE_IS_IDLE then
  START_MACHINE_UP;
else
  COUNT_TIME_DOWN ( CURRENT_TIME );
end if;
```

Control Statements

Conditional

IF

```
if MY_VALUE = 27 then
  HIS_VALUE := 21;
  THEIR_VALUE := 22;
end if;

if MACHINE_IS_RUNNING then
  SET_NEW_SPEED ( 47 );
else
  COUNT_TIME_DOWN ( CURRENT_TIME );
end if;
```

Control Statements

Sequential

BLOCK

-- Used to localize declarations and/or effects

```
procedure MAIN_PROGRAM is
  VARIABLE : FLOAT;
begin
   -- some statements
   declare
      LOCAL_VARIABLE : FLOAT;
   begin
      LOCAL_VARIABLE := 4.0;
      VARIABLE := 70.0;
   end;
   VARIABLE := 10.0;
end MAIN_PROGRAM;
```

Control Statements

Sequential

RETURN

-- For a function, returns a value

```
function IS_GREATER ( FIRST, SECOND : in INTEGER )
                          return BOOLEAN  is
begin
   return ( FIRST > SECOND );
end IS_GREATER;
```

-- Every function must have at least one return statement

Control Statements

Sequential

RETURN

-- Causes control to be passed back to the caller of a subprogram

For a procedure...

```
procedure A_PROCEDURE is
   AN_INTEGER : INTEGER;
begin
   AN_INTEGER := 5;
   return;
   null; -- never gets executed
end A_PROCEDURE;
```

Control Statements

Sequential

ASSIGNMENT

-- Replaces variable on left with expression on right

```
AN_INTEGER := ( 5*2) + 34;
```

PROCEDURE CALL

-- Executes a procedure

```
POP ( AN_INTEGER, OFF_OF => MY_STACK );
```

NULL

-- Explicitly does nothing

```
null;
```

Control Statements

SEQUENTIAL	CONDITIONAL	ITERATIVE
ASSIGNMENT	IF	LOOP
PROCEDURE CALL	CASE	
RETURN		
NULL		
BLOCK		

TASKING	OTHERS
ENTRY CALL	GOTO
DELAY	RAISE
ABORT	CODE
ACCEPT	
SELECT	

Types

Private types

```
with INTEGER_STACK;
use INTEGER_STACK;
procedure STACK_THEM is
     MY_STACK,YOUR_STACK:STACK;
     AN_ITEM: INTEGER
begin
     PUSH (ITEM=>20,ON=>MY_STACK);
     PUSH(ITEM=>30,ON=>YOUR_STACK);
     PUSH(40,ON=>MY_STACK);
     .
     .
     POP(AN_ITEM,OFF_OF=>MY_STACK);
     --AN_ITEM = 40
end STACK_THEM;
```

Types

Private types

```
package INTEGER_STACK is
    type STACK is limited private;
    procedure POP ( ITEM : out INTEGER;
                    OFF_OF:in out STACK);
    procedure PUSH (ITEM: in INTEGER;
                    ON: in out STACK);
private
    --Define what a stack looks like
end INTEGER_STACK;
```

Types

```
with B_R; use B_R;
procedure ICE_CREAM is

   YOUR_NUMBER : NUMBERS;
   procedure GO_TO_DQ is separate;

begin
   TAKE ( YOUR_NUMBER );
   loop
      if NOW_SERVING = YOUR_NUMBER then
         SERVE ( YOUR_NUMBER );
         exit;
      else
         GO_TO_DQ;
         exit;
      end if;
   end loop;

end ICE_CREAM;
```

Types

```
package B_R is

   type NUMBERS is limited private;

   procedure TAKE ( A_NUMBER : out NUMBERS );
   function NOW_SERVING return NUMBERS;
   procedure SERVE ( NUMBER : in NUMBERS );
   function "=" ( LEFT, RIGHT : in NUMBERS ) return
                                          BOOLEAN;

private

   type NUMBERS is range 0..99;

end B_R;
```

Types

```
with B_R; use B_R;
procedure ICE_CREAM is

  YOUR_NUMBER : NUMBERS;

begin

  TAKE ( YOUR_NUMBER );
  loop

    if NOW_SERVING = YOUR_NUMBER then
      SERVE ( YOUR_NUMBER );
      exit;
    else
      YOUR_NUMBER := NOW_SERVING;
    end if;
  end loop;

end ICE_CREAM;
```

Types

```
package B_R is

   type NUMBERS is private;

   procedure TAKE ( A_NUMBER : out NUMBERS );
   function NOW_SERVING return NUMBERS;
   procedure SERVE ( NUMBER : in NUMBERS );

private

   type NUMBERS is range 0..99;

end B_R;
```

Types

```
with B_R; use B_R;
procedure ICE_CREAM is

  YOUR_NUMBER : NUMBERS;

begin

  TAKE ( YOUR_NUMBER );
  loop
    if NOW_SERVING = YOUR_NUMBER then
      SERVE ( YOUR_NUMBER );
      exit;
    else
      YOUR_NUMBER := YOUR_NUMBER - 1;
    end if;
  end loop;

end ICE_CREAM;
```

Ada Generics

by

.Cdr Lindy Moran
USNA

Maj Chuck Engle
SEI

Part of the
Advanced Ada Workshop
sponsored by the
Ada Software Engineering Education
& Training Team (ASEET)

Acknowledgement:

Much of the content of this tutorial comes from an earlier ASEET Advanced Ada Workshop tutorial on generics presented by Mr. John Bailey. Many thanks are due for the interesting examples and ideas!

GENERICS

- ☐ Why program at all?
- ☐ Why program generically?
- ☐ What does generics provide?
- ☐ How do you write a generic unit?
  - ☐ Parameterless Generics
  - ☐ Parameterized Generics
    - ☐ Value and Object Parameters
    - ☐ Type Parameters
    - ☐ Subprogram Parameters
- ☐ What are the Cons of generics?
- ☐ What are the Pros of generics?
- ☐ What are the unresolved issues?
- ☐ How do you teach generics?

Why program at all?

- ☐ Reusability - a programmed solution can be used over and over

- ☐ Reliability - program can be tested and verified to ensure correct results for subsequent runs

- ☐ Readability - program formalizes human solution and represents it in more abstract readable form

- ☐ Maintainability - making a change to a program ensures that the change is consistently applied to all problem solutions

Why program generically?

- ☐ Reusability - similar program units needed but different enough to preclude simply entering differing values at run time

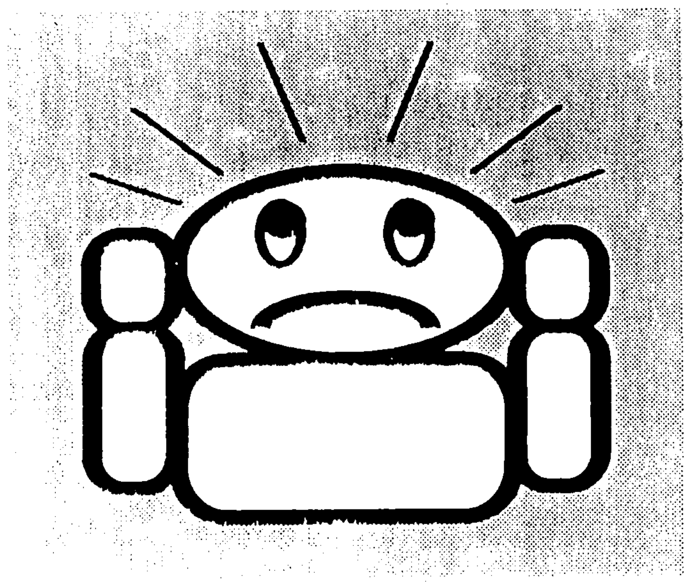
- ☐ Reliability - generic unit once tested and verified does not need to be retested for each new use or "instantiation"

- ☐ Readability - using generic unit allows extraction of the "essence" of the unit eliminating application specific details and produces a very uncluttered readable unit

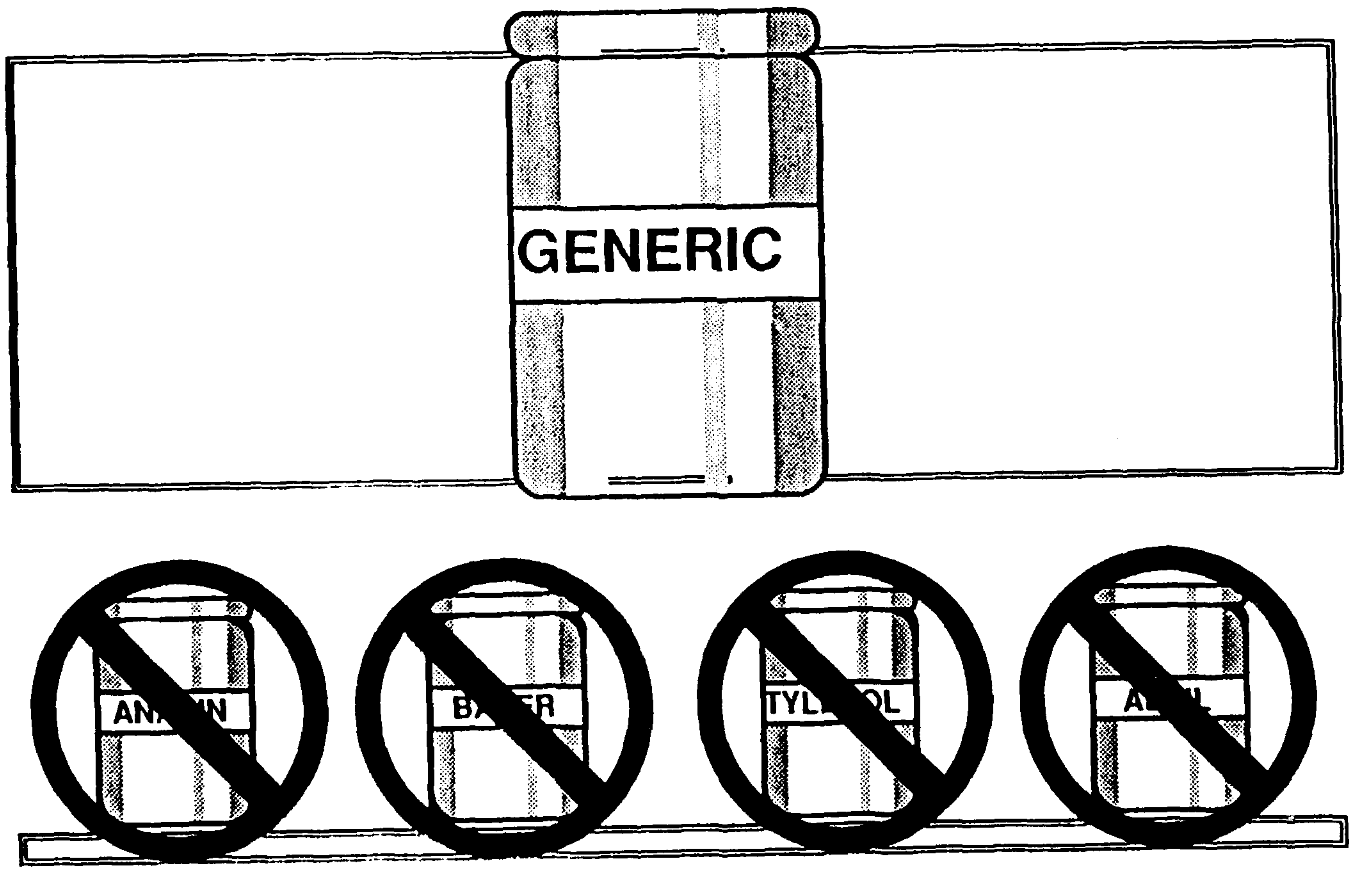
- ☐ Maintainability - a change made to the unit applies to all uses of the unit

- ☐ Programming in the large - facilitates concentration on higher layers of abstraction by providing reusable conceptual building blocks

Strong typing giving you a heADAche?



Try Generic Templates

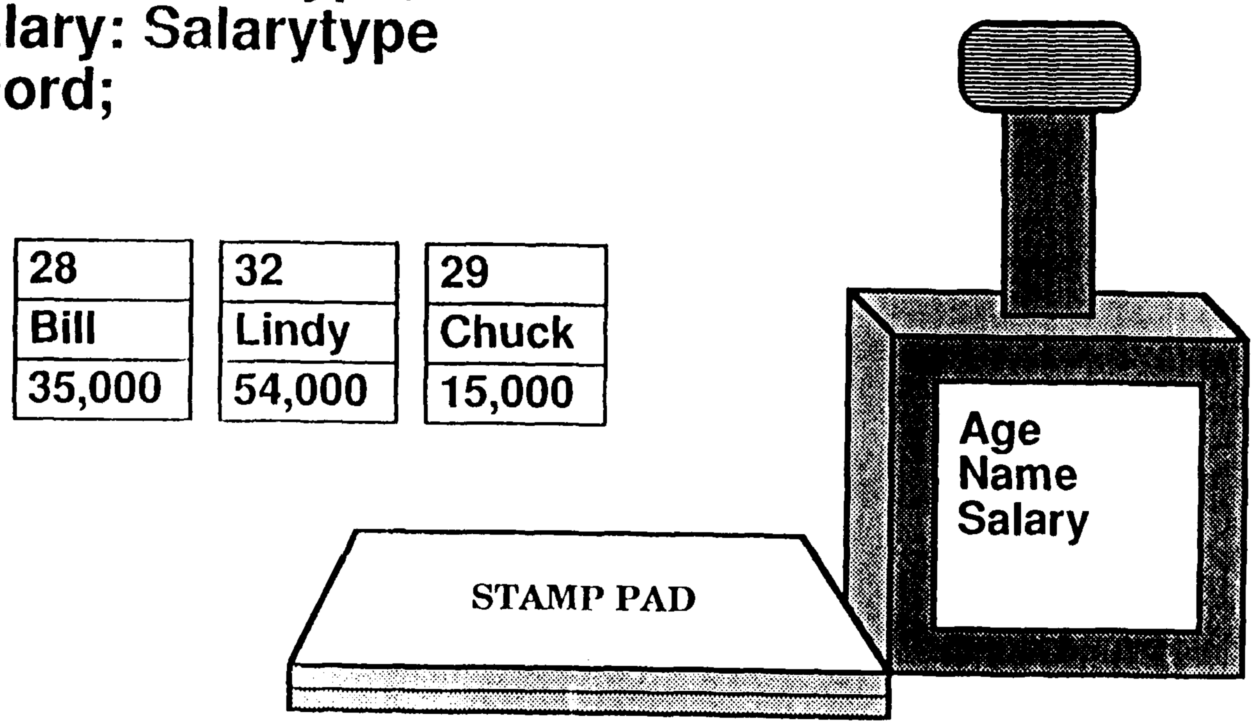


What does generics provide?

- Templates for conceptual building blocks
- Remove problem specifics => greater clarity and understandability of code
- Can add levels of abstraction
- Reduces source code size => code more readable and maintainable
- Facilitates REUSE of software
- Elegant complement to strong typing
- Mechanism for doing I/O

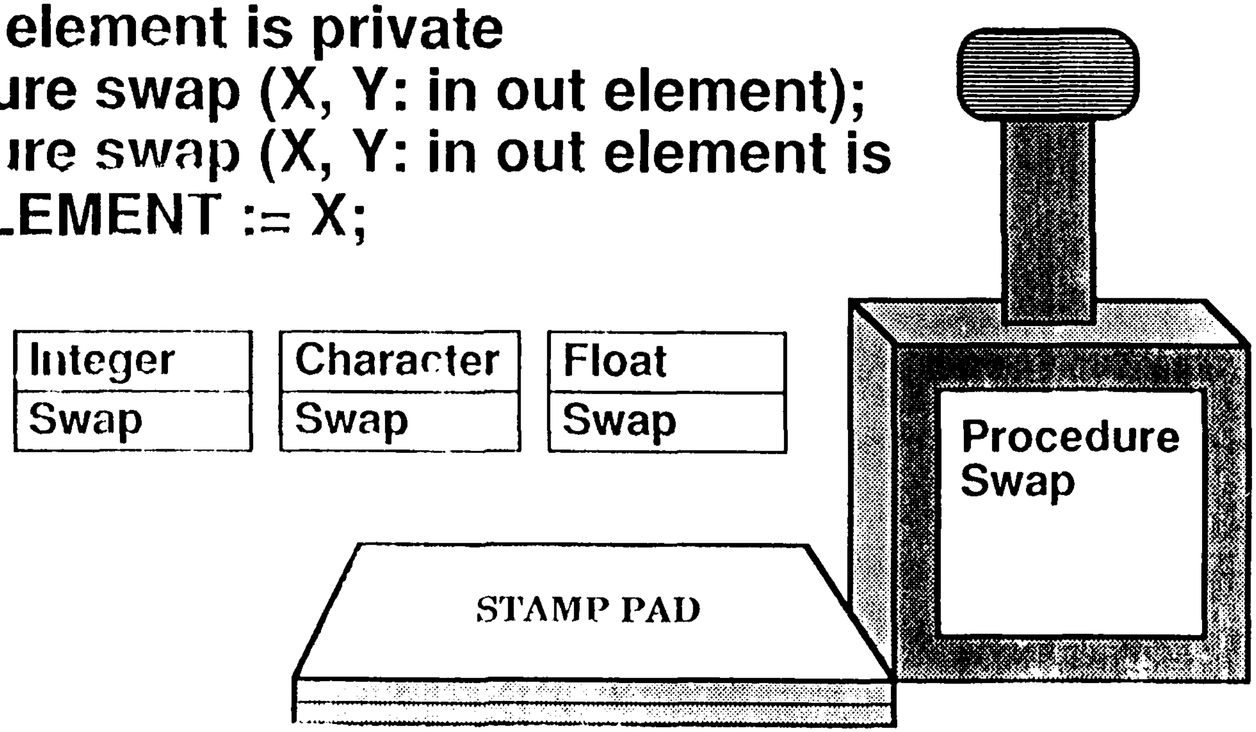
type $\Rightarrow$ template for object

```
type person is record
      Age: Agetype;
      Name: nametype;
      Salary: Salarytype
end record;
```

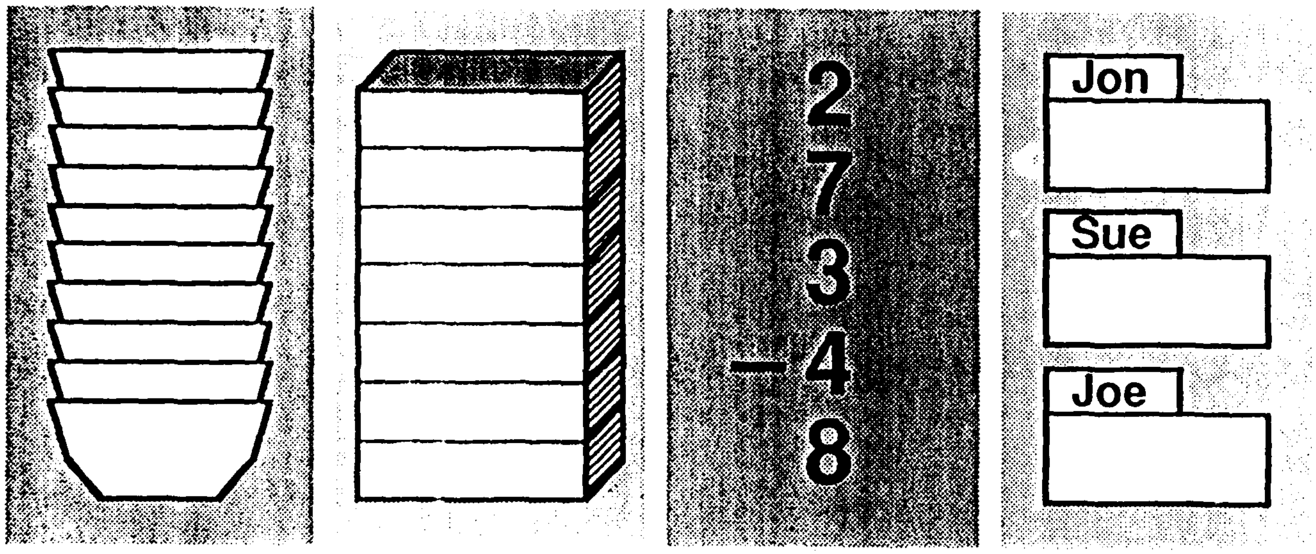


generic ⇒ template for package, function, procedure

```
generic
   type element is private
procedure swap (X, Y: in out element);
procedure swap (X, Y: in out element is
   T: ELEMENT := X;
begin
   .
   .
   .
end;
```



Generic Stack Packages



stack of bowls

stack of books

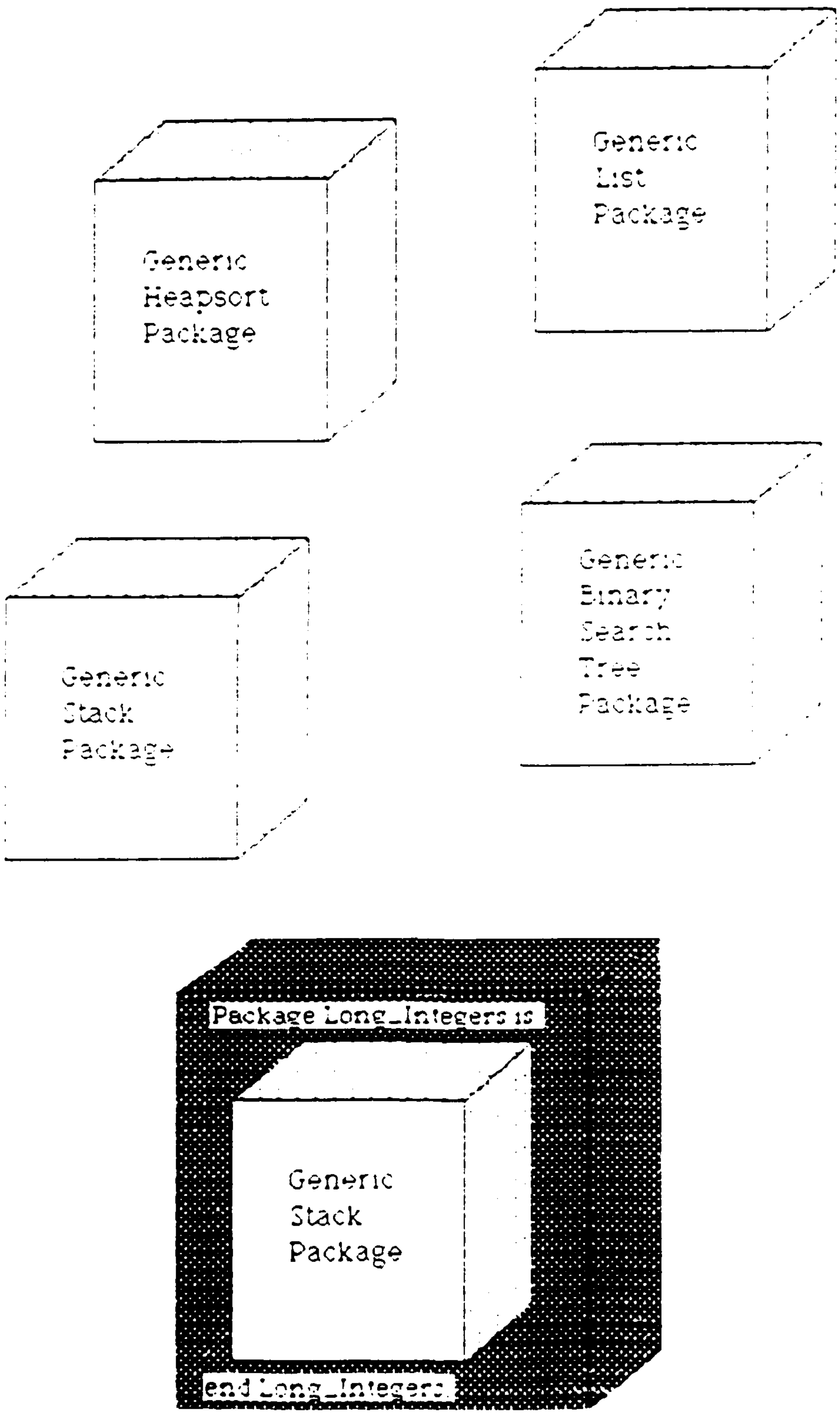
stack of integers

stack of person records

Creating a "Need" for Generics
- A Simple Example -

- Long Integers Problem
  - Problem is to be able to add and multiply non-negative integers of unlimited digits
  - Simple problem to understand
  - Creates "cognitive dissonance" and "need" in student to solve problem
  - Need for generic unbounded stack is relatively obvious
  - Illustrates layers of abstraction
    - Long Integer - Top Level
      - Original level of student focus
    - Stack - Bottom Level
      - Second level of student focus

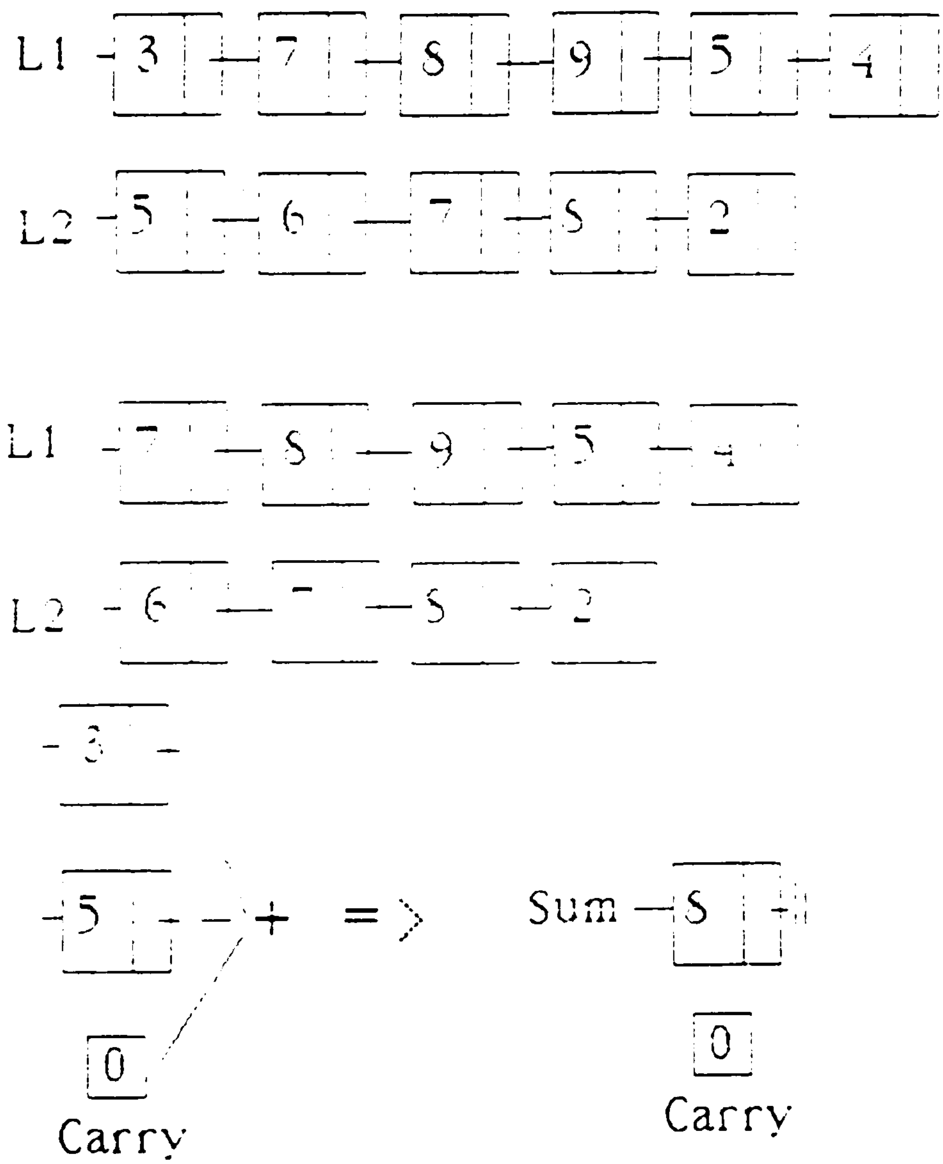
Conceptual Building Blocks



Long Integers Problem

An Example:

```
  459873   (L1)
+  28765   (L2)
---------
  488638   (Sum)
```



```
with Long_Integer_Stack;
package Long_Integers is

   type Long_Integer is private;

   function Make_Long_Integer(Numeral : in string) return Long_Integer;

   function "+"(First_Long_Integer, Second_Long_Integer : Long_Integer)
      return Long_Integer;

   function "*"(N : Natural; A_Long_Integer : Long_Integer)
      return Long_Integer;

   function "*"(First_Long_Integer, Second_Long_Integer : Long_Integer)
      return Long_Integer;

   procedure Put(A_Long_Integer : in Long_Integer);

private
   type Long_Integer is new Long_Integer_Stack.Stack;
end Long_Integers;
```

```
th Text_IO;
ckage body Long_Integers is

 use Long_Integer_Stack;

 function Make_Long_Integer(Numeral : in string) return Long_Integer is
    L : Long_Integer;
 begin
    Clear(L);
    for Position in Numeral'first..Numeral'last loop
       Push(character'pos(Numeral(Position))-character'pos('0'),L);
    end loop;
    return L;
 end Make_Long_Integer;

 function "+"(First_Long_Integer, Second_Long_Integer : Long_Integer)
    return Long_Integer is
    ReversedSum, Sum : Long_Integer;
    Carry : integer := 0;
    SingleColumnSum : integer := 0;
    L1 : Long_Integer := First_Long_Integer;
    L2 : Long_Integer := Second_Long_Integer;
 begin

    Clear(ReversedSum);
    Clear(Sum);

    while (NOT Is_Empty(L1)) and (NOT Is_Empty(L2)) loop
       SingleColumnSum := Top_Of(L1) + Top_Of(L2) + Carry;
       Push(SingleColumnSum mod 10,ReversedSum);
       Carry := (SingleColumnSum - (SingleColumnSum mod 10)) / 10;
       Pop(L1);
       Pop(L2);
    end loop;

    while NOT Is_Empty(L1) loop
       SingleColumnSum := Top_Of(L1) + Carry;
       Push(SingleColumnSum mod 10,ReversedSum);
       Carry := (SingleColumnSum - (SingleColumnSum mod 10)) / 10;
       Pop(L1);
    end loop;

    while NOT Is_Empty(L2) loop
       SingleColumnSum := Top_Of(L2) + Carry;
       Push(SingleColumnSum mod 10,ReversedSum);
       Carry := (SingleColumnSum - (SingleColumnSum mod 10)) / 10;
       Pop(L2);
    end loop;

    if Carry = 1 then
       Push(1,ReversedSum);
    end if;

    while NOT Is_Empty(ReversedSum) loop
       Push(Top_Of(ReversedSum),Sum);
       Pop(ReversedSum);
    end loop;

    return Sum;
 end "+";
```

```
      for Count in 1..N loop
         Result := Result + A_Long_Integer;
      end loop;
      return Result;
   end "*";

   function "*"(First_Long_Integer, Second_Long_Integer : Long_Integer)
      return Long_Integer is
      L1 : Long_Integer := First_Long_Integer;
      L2 : Long_Integer := Second_Long_Integer;
      Result : Long_Integer := Make_Long_Integer("0");
      Digit : integer;
      Position : integer := 0;
      Temp : Long_Integer;
   begin
      while NOT Is_Empty(L1) loop
         Digit := Top_Of(L1);
         Pop(L1);
         Position := Position + 1;
         Temp := Digit * L2;
         for NumberOfTrailingZeros in 2..Position loop
            Push(0,Temp);
         end loop;
         Result := Result + Temp;
      end loop;
      return Result;
   end "*";

   procedure Put(A_Long_Integer : in Long_Integer) is
      Temp, Temp2 : Long_Integer;
   begin
      Temp := A_Long_Integer;

      -- reverse contents of Temp into Temp2
      while NOT Is_Empty(Temp) loop
         Push(Top_Of(Temp),Temp2);
         Pop(Temp);
      end loop;

      -- print contents of Temp2 on screen
      while NOT Is_Empty(Temp2) loop
         Text_IO.Put(integer'image(Top_Of(Temp2))(2));
         Pop(Temp2);
      end loop;
   end Put;

end Long_Integers;
```

```
:h Long_Integers, Text_IO; use Long_Integers, Text_IO;
ocedure Uselongintegers is
 A, B : Long_Integer;
gin
 A := Make_Long_Integer("25012345");
 B := Make_Long_Integer("22334455");
 Put(A * B);
 New_Line;
 Put(2*A);
i UseLongIntegers;
```

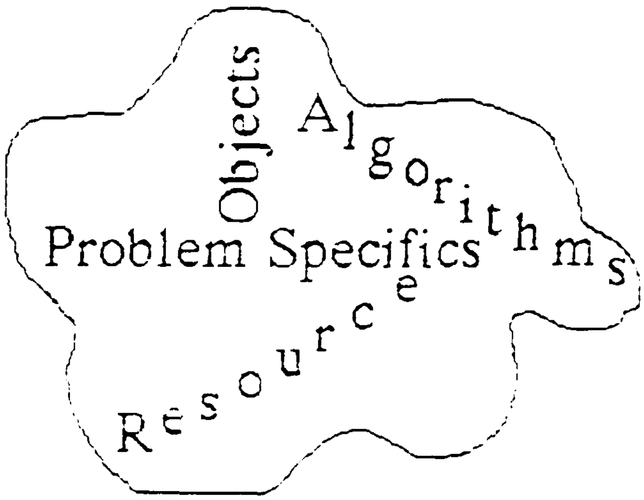
```
generic
    type Item is private;
package Stack_Sequential_Unbounded_Unmanaged_Noniterator is

    type Stack is limited private;

    procedure Copy  (From_The_Stack : in     Stack;
                      To_The_Stack   : in out Stack);
    procedure Clear (The_Stack      : in out Stack);
    procedure Push  (The_Item       : in     Item;
                      On_The_Stack   : in out Stack);
    procedure Pop   (The_Stack      : in out Stack);

    function Is_Equal (Left      : in Stack;
                       Right     : in Stack) return Boolean;
    function Depth_Of (The_Stack : in Stack) return Natural;
    function Is_Empty (The_Stack : in Stack) return Boolean;
    function Top_Of   (The_Stack : in Stack) return Item;

    Overflow  : exception;
    Underflow : exception;

private
    type Node;
    type Stack is access Node;
end Stack_Sequential_Unbounded_Unmanaged_Noniterator;
```

[Taken from Software Components with Ada by Grady Booch]

```
th Stack_Sequential_Unbounded_Unmanaged_Noniterator;
ckage Long_Integer_Stack is new
 Stack_Sequential_Unbounded_Unmanaged_Noniterator(Item=>integer);
```

Traditional Programming

Algorithms, Objects, Resources
-- intermixed with --
Problem specifics



```
procedure Swap(X,Y : in out integer)         is
  Temp : integer          := X;
begin
  X := Y;
  Y := Temp;
end;

procedure Swap(X,Y : in out character)       is
  Temp : character        := X;
begin
  X := Y;
  Y := Temp;
end;

procedure Swap(X,Y : in out float)           is
  Temp : float            := X;
begin
  X := Y;
  Y := Temp;
end;


type AnArray is array(1..10) of integer;

procedure Swap(X,Y : in out AnArray)         is
  Temp : AnArray          := X;
begin
  X := Y;
  Y := Temp;
end;
```

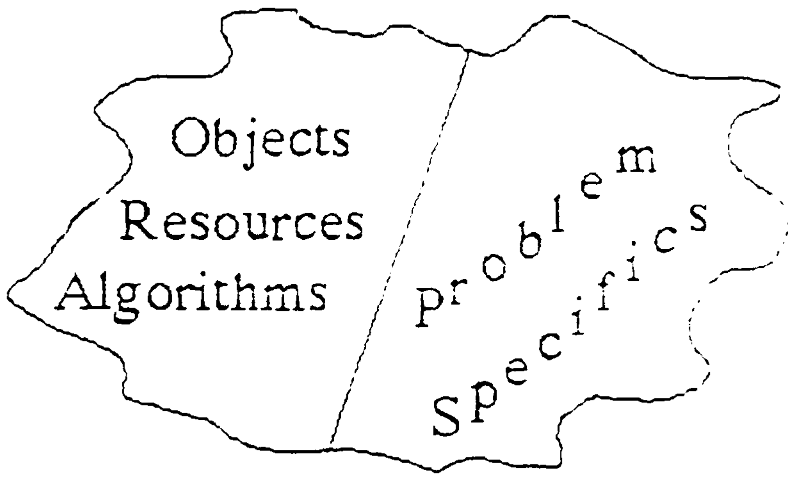
Generic Programming

Algorithms, Objects, Resources

---

separated from

Problem specifics



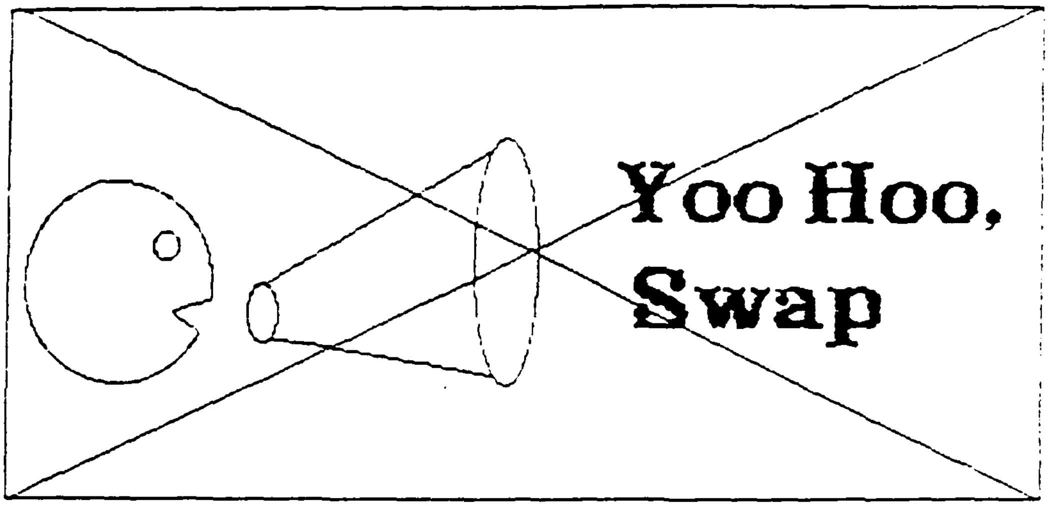
Syntax and Semantics

generic
. . . generic formal parameters . . .
subprogram or package specification;

subprogram or package body

A Generic Swap Procedure

```
generic
  type Element is private;
procedure Swap(X,Y : in out Element);

procedure Swap(X,Y : in out Element) is
  Temp : constant Element := X;
begin
  X := Y;
  Y := Temp;
end Swap;
```



NO!! Generic units not "callable/usable"!!

Explicit Instantiation

- Creates callable/usable unit

```
with Swap;
procedure Example is
  . . .
  procedure CharSwap is new Swap(character);
  procedure IntSwap is new Swap(Element=>integer);
  . . .
begin
  . . .
  CharSwap(OneLetter,AnotherLetter);
  IntSwap(AnInteger, AnotherInteger);
  . . .
end Example;
```

Overloading Instance Names

```
with Swap;
procedure SwapThings is
   X : integer := 5;
   Y : integer := 10;
   A : character := 'A';
   B  : character := 'B';

  procedure Exchange is new Swap(character);
  procedure Exchange is new Swap(integer);

begin
  Exchange(X,Y);
  Exchange(A,B);
end;
```

Generic Units
An Analogy

	Declaration	Instantiation
Data Object:	Type Declaration	Object Declaration
	type Age is range 0..100;	*OldAge : Age ;*
Generic Unit:	Generic Declaration	Generic Instantiation
	generic *type Element is private;* *procedure DoSomething;* *procedure DoSomething is* *X : Element;* *begin* *... do something ...* *end DoSomething;*	*procedure DoThis is* *new DoSomething* *(Element=>integer);*

Explicit Instantiation

```
generic
   type Element is <>;
procedure Swap (X,Y : in out Element);
procedure Swap (X,Y : in out Element) is
   Temp : Element := X;
begin
   X := Y;
   Y := Temp;
end;

with Swap;
procedure SwapThings is
   X : integer := 5;
   Y : integer := 10;
   A : character := 'A';
   B : character := 'B';
begin
   Swap(X,Y);    -- Why NOT?
   Swap(A,B);    -- param types differ after all
end SwapThings;
```

- ☐ Requirement to EXPLICITLY instantiate simplifies compilation of units

- ☐ The explicit instantiation provides well-defined locus for reporting errors arising from inconsistent substitutions

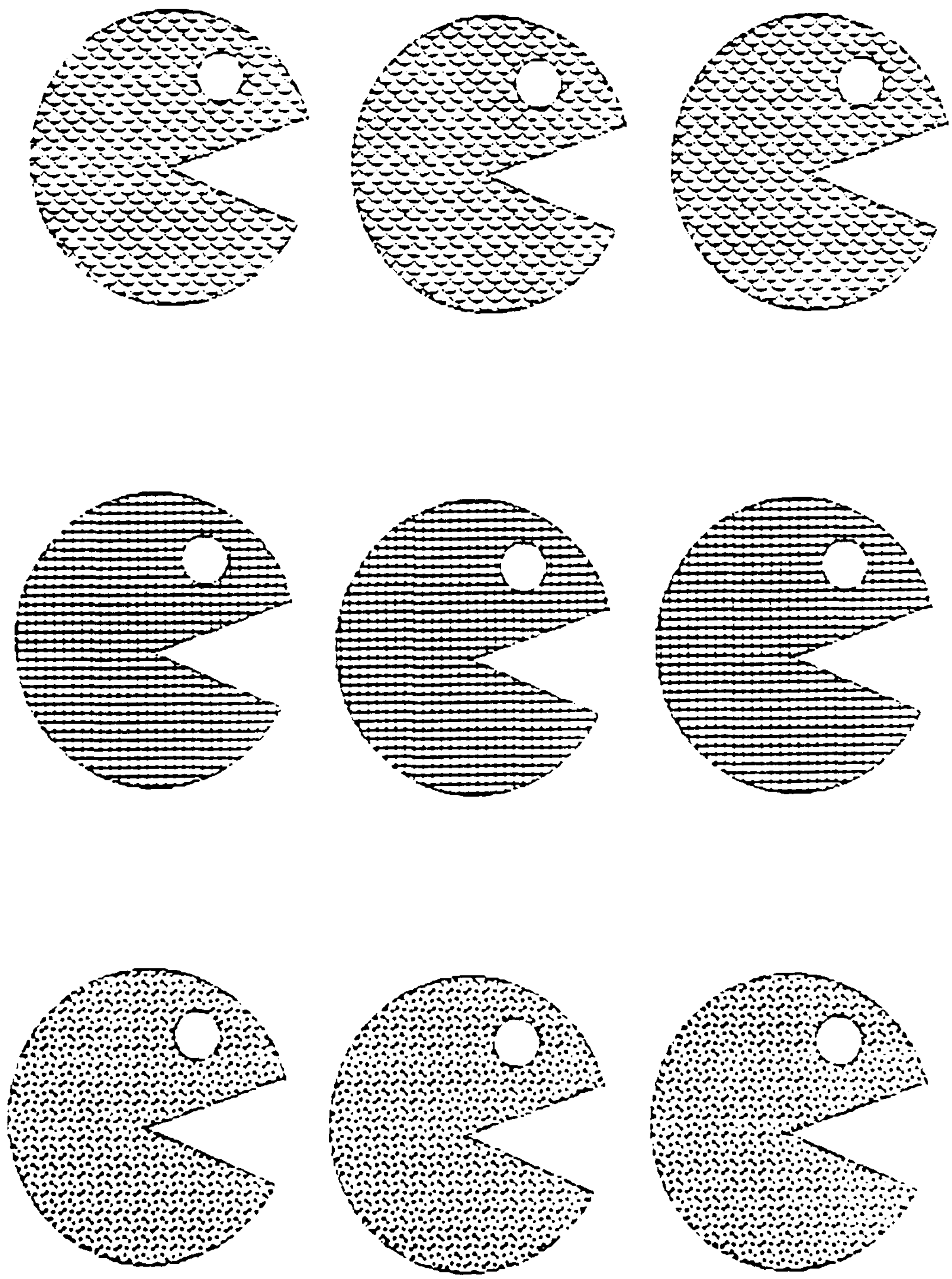
Explicit Instantiation (continued)

- Permits independent checking of generic units and generic instantiations
- Resolves ambiguity of reference that might otherwise occur
- Provides better awareness of instances and improves reliability and readability

```
with Swap;
procedure SwapThings is
  X : integer := 5;
  Y : integer := 10;
  A : character := 'A';
  B : character := 'B';

  procedure Swap(X,Y : in out character) is
  begin
    X := 1;
    Y := 1;
  end Swap;

begin
  Swap(X,Y);    -- generic Swap used
  Swap(A,B);    -- local Swap masks generic one
end SwapThings;
```

- What about recursive calls in the generic

"Cloning" Things



Parameterless Generics
"Cloning" Units

A nongeneric "unique object" Stack package:

```
package Stack is
  procedure Pop(I : out integer);
  procedure Push(I : in integer);
  function Empty return boolean;
  function Full return boolean;
end Stack;
```

A non-generic "many objects" solution:

```
package Stacks is
  type Stack is . . .;
  procedure Pop(S : in out Stack; I : out integer);
  procedure Push(S : in out Stack; I : in integer);
  function Empty(S : Stack) return boolean;
  function Full(S : Stack) return boolean;
end Stacks;

-- changes must be made to body of package also
```

A sample user program:

```
procedure StackUp is
  S1, S2 : Stack;   Item : integer;
begin
  Push(S1,10); Push(S2,5); Pop(S1,Item);
end;
```

Parameterless Generics cont.

A generic "many objects" solution:

```
generic
package Stack is
  procedure Pop(I : out integer);
  procedure Push(I : in integer);
  function Empty return boolean;
  function Full return boolean;
end Stack;

-- generic body is identical to non-generic one
-- no changes have to be made to get many stacks
```

A sample user program:

```
with Stack;
procedure StackUp is
   Item : integer;
   package S1 is new Stack;
   package S2 is new Stack;
begin
  S1.Push(10);  S2.Push(5);
  S1.Pop(Item);  S2.Pop(Item);
end StackUp;
```

Parameterless Generics cont.

- Stack implementations compared
  - Non-generic package - only one elaboration and initialization occur
  - Generic package - multiple elaborations and initializations occur - once for each package

Example:
```
with Text_IO;
package body Stack is
  ...
begin
  Text_IO.Put("New stack created.");
end Stack;

package S1 is new Stack; -- message prints
package S2 is new Stack; -- message prints again
pacakge S3 is new Stack; -- message prints again
. . .
```

Creating Library Units of Generic Instantiations

```
-- compile following separately into the library

     with Stack;
     package S1 is new Stack;


-- S1 is now a usable library unit

with S1; use S1;
procedure StackUp is
   Item : integer;
begin
   Push(10);
   Push(20);
   Pop(Item);
end StackUp;
```

Parameterized Generics

- Generic Parameters
  - Value and Object Parameters
  - Type Parameters
  - Subprogram Parameters

Value and Object Parameters

- Value Parameters
  - Are of mode IN
  - Serve as local constants in generic units
- Object Parameters
  - Are of mode IN OUT
  - Serve as global objects in generic units

Value Parameters

```
generic
  Max : in integer;
  Min : integer;   -- default mode is IN
procedure BigNSmall(X : in integer);

procedure BigNSmall(X : in integer) is
begin
   if X > Max then
     Max := X;          -- not with mode IN
   end if;
   if X < Min then
     Min := X;          -- not with mode IN
   end if;
end BigNSmall;
```

Value Parameters
and
Initialization Before Instantiation

- ☐ Actual parameters which are to match with formal generic value parameters "must" have been initialized before the instantiation occurs

Example:

```
generic
    Max : in integer;
    Min : integer;   -- default mode is IN
procedure BigNSmall(X : in integer);

with BigNSmall;
procedure UseBigNSmall is
   LocalMin : integer;         --no initial value
   LocalMax : integer;         -- no initial value
   X : integer :=100;

   procedure Extremes is new
        BigNSmall(Max=>LocalMax,Min=>LocalMin);
  -- run-timeerror occurs due to lack of initialization IF contents
  -- of uninitialized objects raises constraint_error
begin
   Extremes(X);
end UseBigNSmall;
```

Value Parameters
and
Levels of Abstraction

```
generic
   Lower, Upper : in character;
function In_Range(S : in string) return boolean;

function In_Range(S : in string) return boolean is
begin
  for I in S'Range loop
    if S(I) not in Lower..Upper then
      return FALSE;
    end if;
  end loop;
  return TRUE;
end In_Range;
```

A non-generic version of In_Range:

```
function In_Range(S : in string; Upper,Lower :
      character) return boolean is
begin
  for I in S'Range loop
    if S(I) not in Lower .. Upper then
      return FALSE;
    end if;
  end loop;
  return TRUE;
end In_Range;
```

Value Parameters and Levels of Abstraction cont.

- Compare clarity in user's programs using generics to add another level of abstraction in "customized" names for In_Range function

```
with In_Range;
procedure InBounds is
  Name : string(1..4) := "JACK";
  Phone : string(1..7) := "6725643";
begin
  if In_Range(Name,'A','Z') then ...
  if In_Range(Phone,'0','9') then ...
end InBounds;
```

```
with In_Range;
procedure InBounds is
  Name : string(1..4) := "JACK";
  Phone : string(1..7) := "6725643";

  function Is_All_Upper_Case is new In_Range('A','Z');

  function Is_All_Lower_Case is new In_Range('a','z');

  function Is_All_Decimal is new In_Range('0','9');

begin
  if Is_All_Upper_Case(Name) then ...
  if Is_All_Decimal(Phone) then ...
end InBounds;
```

[*In_Range taken from Ada Language and Methodology]

Value Parameters

Our Stack Example Revisited

```
generic
  Size : in natural;
package Stacks is
  type Stack is limited private;
  procedure Push(S : in out Stack; I : in integer);
  procedure Pop(S : in out Stack; I : out integer);
private
  subtype NumberOfElements is integer
    range 0..Size;
  type ElementArray is
    array(NumberOfElements) of integer;
  type Stack is record
    Elements : Element_Array;
    Top : NumberOfElements := 0;
  end record;
end Stacks;

with Stacks;
procedure StackUp is
  package SmallStack is new Stacks(5);
  pacakge BigStack is new Stack(5000);
begin
   . . .
end StackUp;
```

Value Parameters and Default Values

(only on VALUE parameters, not OBJECT parameters)

```
generic
  Rows : in positive := 24;
  Columns : in positive :- 80;
package Terminal is
 . . .
end Terminal;

-- some possible instantiations

package MicroTerminal is new Terminal(24,40);
-- using positional notation

package WordProcessor is new
   Terminal(Columns=>85,Rows=>66);
-- using named notation

package DefaultTerminal is new Terminal;
-- using the default values of 24 and 80

package NewTerminal is new
   Terminal(X+Y,Z+10);
-- using expressions
```

Value Parameters and The Subtleties of Default Values

What are the outputs of the following?

```
package CountingPackage is
  function NextNum return integer;

  generic
    Val : integer := NextNum;
  procedure Count;
end CountingPackage;

with Text_IO;
package body CountingPackage is
  CurrentValue : integer := 0;
  function NextNum return integer is
  begin
    CurrentValue := CurrentValue + 1;
    return CurrentValue;
  end NextNum;

  procedure Count is
  begin
    Text_IO.Put_Line(integer'image(Val));
  end Count;
end CountingPackage;

with CountingPackage;
procedure StartCounting is
  procedure FirstCount is new CountingPackage.Count;
  procedure CountAgain is new CountingPackage.Count;
begin
  FirstCount;
  CountAgain;
end StartCounting;
```

AN IMPLEMENTATION DEPENDENCY

```
with Text_IO; use Text_IO;
procedure Imp is

  Counter : integer := 0;

  generic
     A : in integer;
     B : in integer;
  procedure X;

  procedure X is
  begin
     put_line(integer'image(A+B));
  end X;

  function Next return integer is
  begin
     Counter := Counter + 1;
     return Counter;
  end Next;

  procedure InstanceOfX is new X([redacted]);

begin
   InstanceOfX;
end Imp;
```

order of evaluation is implementation dependent

Value Parameters and Limited Types

- ☐ Value parameters are constants whose value is a copy of the value of the generic actual parameter supplied in the instantiation.

- ☐ Type of generic formal value parameter therefore cannot be limited type because copy of actual parameter value cannot be assigned to it.

```
with Text_IO;
generic
  MyFile : Text_IO.File_Type;  -- NO!
procedure Wrong;

-- problem is File_Type is limited private
```

Object Parameters

A More Useful Example

```
generic
  Control_Block : in out DeviceData;
  Kind : in VDU_Kind := Basic_Kind;
package VDU is
  . . .
end VDU;

with VDU;
procedure ManyVDUs is
  DeviceTable : array(1..N) of DeviceData;

  package VDU1 is new
     VDU(DeviceTable(1),Kind_A);
  package VDU2 is new
     VDU(DeviceTable(2),Kind_B);

begin
  . . .
end ManyVDUs;
```

[*Taken from Ada Language and Methodology by Watt, Wichmann, and Findlay]

Object Parameters and Subtleties

- Object parameters passed by reference not by copy-restore method

- Object parameters are "aliases" for their actual parameter counterparts

Example:

```
with Text_IO; use Text_IO;
procedure X is
  Global : integer := 99;
  procedure Z(Param : in out integer) is
  begin
    Param := Param + 1;
    Put_Line(integer'image(Param));
    Put_Line(integer'image(Global));
  end Z;
begin
  Z(Global);
end X;

-- output is 100, 99 for copy-restore method
-- output is 100,100 for pass by reference
```

Object Parameters and Subtleties cont.

- Object parameters passed by reference not by name -- not like Algol's "copy rule"

- Address of actual parameter corresponding to formal generic object parameter is evaluated ONCE and does not change

- Using generic object parameter NOT like doing textual substitution of actual parameter's name

Object Parameters and Subtleties cont.

- ADDRESS of actual parameter corresponding to a generic formal object parameter is evaluated at time of instantiation

```
declare
  Y : array(1..5) of character := "kitty";
  Index : integer := 1;

  generic
    X : in out character;
  procedure Replace;

  procedure Replace is
  begin
    Index := 5;
    X := 'w';              -- X => Y(1), NOT Y(5)
    Put(String(Y));
  end Replace;

  procedure Update is new Replace(Y(Index));
  -- Index = 1 when this instantiation occurs

begin
   Update;
end;
```

NON-EXAMPLE

```
declare
  Y : array(1..5) of character := "kitty";
  Index : integer := 1;

  generic
    X : in out character;
  procedure Replace;

  procedure Replace is
  begin
    Index := 5;
    Y(Index) := 'w';
    Put(String(Y));
  end Replace;

 procedure Update is new Replace(Y(Index));
  -- Index = 1 when this instantiation occurs

begin
  Update;
end;
```

```
declare
  subtype Small is integer range 1 .. 10;
  X : integer := 27;
  generic
    S : in Small;
  procedure Gen;
  procedure Gen is
  begin
     Put("All OK");
  end Gen;
  procedure P is new Gen(X);
  -- Constraint_Error raised at time of instant.
begin
   P;
end;
```

```
declare
  subtype Small is integer range 1..10;
  X : integer := 27;
  generic
    S : in out Small;
  procedure Gen;
  procedure Gen is
  begin
     Put("All OK");
  end Gen;
  procedure P is new Gen(X);
  -- executes OK --
begin
  P;
end;
```

Object Parameters and Constraints Imposed

- Constraints applied to generic formal object parameter are those of corresp. ACTUAL parameter.

```
declare
 subtype Small is integer range 1..10;
 X : integer := 10;

 generic
  S : in out Small;
 procedure Constraints;
 procedure Constraints is
 begin
  S := S + 1;
 end;

 procedure ActualConstraint is new
   Constraints(X);  -- causes NO problem
                    -- constraints of integer apply
begin
  ActualConstraint;
end;
```

```
declare
 subtype Small is integer range 1..10;
 X : Small := 10;

 generic
  S : in out Small;
 procedure Constraints;
 procedure Constraints is
 begin
  S := S + 1;
 end;

 procedure ActualConstraint is new
   Constraints(X); -- causes problem
                   -- constrains of Small apply
begin
  ActualConstraint;
end;
```

Object Parameters

- ☐ Use not recommended because suffer from all same falacies as global objects

- ☐ Generic object parameters usually SHOULD have been regular formal parameters in the subprogram

Object Parameters cont.

```
generic
  Variable : in out integer;
  Limit, ResetValue : in integer;
procedure ResetIntegerTemplate;

procedure ResetIntegerTemplate is
begin
  if Variable > Limit then
    Variable := ResetValue;
  end if;
end ResetIntegerTemplate;
```

Better written as . . .

```
generic
  Limit, ResetValue : in integer;
procedure ResetIntegerTemplate(Variable : in out
   integer);

procedure ResetIntegerTemplate(Variable : in out
    integer) is
begin
  if Variable > Limit then
    Variable := ResetValue;
  end if;
end ResetIntegerTemplate;
```

[*Taken from Ada As a Second Language by Cohen]

Object Parameters and Defined Operations

- Operations defined on object are the basic or predefined operators defined for the matching actual type. . . even if operator redefined for actual type or parent type of actual type.

```
with Text_IO; use Text_IO;
procedure NotRedefined is

  function "+"(L,R : integer) return integer is
  begin
    return L+R+1;
  end;

 generic
    type SomeType is range <>;
 function Plus(L,R : SomeType) return SomeType;
 function Plus(L,R : SomeType) return SomeType is
 begin
    return L + R; -- predefined integer plus
 end Plus;

 function PlusInstance is new
    Plus(SomeType=>integer);

begin
  Put_Line(integer'image(PlusInstance(3,4)));
end;
```

Type Parameters

☐ type *identifier* is range <>;

☐ type *identifier* is digits <>;

☐ type *identifier* is delta <>;

☐ type *identifier* is (<>);

☐ type *identifier* is array(*typemark* range <>,
. . . , *typemark* range <>) of *typemark*;

☐ type *identifier* is array(*typemark*, . . . ,
typemark) of *typemark*;

☐ type *identifier* is access *typemark*;

☐ type *identifier* is private;

☐ type *identifier* is limited private;

* no SUBtypes

Integer Type Parameters

- ☐ type *identifier* is range <>;

- ☐ matches an integer type, predefined or user-defined

- ☐ operations defined are those defined for integers such as +,-,/,*,**, rem, mod, negation, abs, >, <, =, /=, <=, >=

- ☐ attributes defined are those defined for integers such as 'first, 'last, 'succ, . . .

Integer Type Parameters
An Example

```
generic
  type IntType is range <>;
function Increment(X : IntType) return IntType;

function Increment(X:IntType) return IntType is
begin
  return X+1;
end Increment;

with Increment;
procedure IncrementThings is

  type Age is range 0 .. 130;
  type Temp is range -100 .. 100;

  MyAge : Age  := 30;
  CurrentTemp : Temp := 50;

  function YearOlder is new Increment(Age);
  function TempUp is new
      Increment(IntType=>Temp);

begin
  MyAge := YearOlder(MyAge);
  CurrentTemp := TempUp(CurrentTemp);
end IncrementThings;
```

Float Type Parameters

- type *identifier* is digits <>;

- matches any floating point type, predefined or user-defined

- operations defined are those available for floating point types such as +, -, /, *, **, negation, abs, >, <, =, /=, <=, >=

- attributes defined are those available for floating point types such as 'small, 'large, 'digits, 'mantisa, 'epsilon, . . .

- useful in providing mathematical routines where user can control the precision used

Float Type Parameters
An Example

```
generic
  type FloatType is digits <>;
function Sqrt(X : FloatType) return FloatType;

function Sqrt(X : FloatType) return FloatType is
begin
  . . .
end Sqrt;

with Sqrt;
procedure Rooting is
   type VeryPrecise is digits 7;
   type Imprecise is digits 3;

   X : VeryPrecise := 0.1234;
   Y : Imprecise := 0.12;

   function ExactRoot is new Sqrt(VeryPrecise);
   function RoundRoot is new Sqrt(Imprecise);

begin
   X := ExactRoot(X);
   Y := RoundRoot(Y);
end Rooting;
```

Discrete Type Parameters

- type *identifier* is (<>);

- matches any discrete type -- includes integer types and enumeration types (boolean also)

- attributes defined are those available for any discrete/scalar type such as 'first, 'last, 'succ, 'pred, 'image, 'value, 'pos, 'val

- operations defined are those defined for discrete/scalar types such as >, <, =, /=, >=, <=

Discrete Type Parameters
An Example

```
generic
  type Element is (<>);
package Sets is
  type Set is private;
  function Intersection(S1,S2 : Set) return Set;
  function Union(S1,S2 : Set) return Set;
  function IsIn(Item : Element; S : Set) return
      boolean;
  function IsNull(S : Set) return boolean;
private
  type Set is array(Element) of boolean;
end Sets;


-- some possible instantiations

package CharacterSet is new Sets(character);

package IntegerSet is new Sets(integer);

type Student is (John, Joan, Ann, Sue, . . ., Zip);
package StudentSet is new Sets(Student);
```

Discrete Type Parameters cont.

- Minimal assumptions about the type must be made - operations must apply to ALL discrete types

Example:

```
generic
  type Element is (<>);
function Next(X : Element) return Element;

function Next(X : Element) return Element is
begin
   X := X + 1;      -- not defined for ALL
                    --   discrete types
end Next;
```

Use attributes:

```
function Next(X : Element) return Element is
begin
   if X = Element'Last then
     return Element'First;
   else
     return Element'Succ(X);
   end if;
end Next;
```

Constrained Array Type Parameters

- type *identifier* is array (*typemark*, . . ., *typemark*) of *typemark*;

- matches any constrained array type where:
  1) number of dimensions match,
  2) index subtypes of corresponding dimensions match,
  3) bounds in corresponding dimensions are identical,
  4) component types match

- attributes defined are those available for constrained arrays such as 'first(n), 'last(n), 'range(n), 'length(n)

- operations defined include those available for constrained arrays such as =, :=, using slice notation (for one dimensional arrays)

Constrained Array Type Parameters
An Example

```
generic
   type Index is range <>;
   type Component is (<>);
   type AnArray is array(Index) of Component;
   -- LRM 12.1.2(2) only discrete range that is
   -- allowed is a type mark...NOT (1..10),etc.
procedure Sort(A : in out AnArray);
procedure Sort(A : in out AnArray) is
  Temp : Component;
begin
  for I in A'first+1 .. A'last loop
    for J in A'first..I-1 loop
      if A(I) < A(J) then
        Temp := A(J);
        A(J) := A(I);
        A(I) := Temp;
      end if;
    end loop;
  end loop;
end Sort;
```

```
-- in user program
subtype Small is integer range 1..10;
type Age is integer range 0..130;
type AgeArray is array(Small) of Age;
X : AgeArray := (8,0,9,4,50,35,87,97,1,124);

  procedure AgeSort is new
      Sort(Index=>Small,
           Component=>Age,
            AnArray=>AgeArray);

... AgeSort(X); ...
```

Unconstrained Array Type Parameters

- type *identifier* is array(*typemark* range <>, . . ., *typemark* range <>) of *typemark*;

- matches any unconstrained array where:
  1) number of dimensions the same
  2) subtype of index for corresponding dimensions is the same
  3) component types match

- attributes defined are those available for unconstrained arrays such as 'first(n), 'last(n), 'range(n), 'length(n)

- operations defined include those available for unconstrained arrays such as =, :=, using slice notation (for one dimensional typearrays)

Unconstrained Array Type Parameters An Example

```
generic
  type Index is range <>;
  type Component is range <>;
  type AnArray is array(Index range <>) of
    Component;
procedure Sort(A : in out AnArray);
procedure Sort(A : in out AnArray) is
  Temp : Component;
begin
  for I in A'First+1 .. A'Last loop
   for J in A'First .. I-1 loop
     if A(I) < A(J) then
        Temp := A(J);
        A(J) := A(I);
        A(I) := Temp;
     end if;
   end loop;
  end loop;
end Sort;
```

```
--in user's program

type Age is range 0..130;
type EmployeeNumber is range 1..100;
type EmpList is array(EmployeeNumber range <>)
   of Age;
procedure EmployeeAgeSort is new
    Sort(Index=>EmployeeNumber,
         Component=>Age,
         AnArray=>EmpList);

   Employees : EmpList(5..50) := (. . . .);


 . . . EmployeeAgeSort(Employees); . . .
```

Private Type Parameters

- type *identifier* is private;

- matches any constrained type except a limited type

- operations available are only declaring objects of the type, testing for equality and inequality, and assigning values to objects of the type

Private Type Parameters
An Example

```
generic
  type Index is (<>);
  type Component is private;
  type AnArray is array(Index) of Component;
function Found(A : AnArray; T : Component)
    return boolean;
function Found(A : AnArray; T : Component)
   return boolean is
begin
  for I in A'First..A'Last loop
    if A(I) = T then
      return TRUE;
    end if;
  end loop;
  return FALSE;
end Found;
```

```
--in user's program

type Student is (Joan,John,Sue,...,Debbie);
type Grade is range 0..100;
type GradeArray is array(Student) of Grade;
function GradeMade is new
   Found(Index=>Student,
         Component=>Grade,
         AnArray=>GradeArray);


Grades : GradeArray := (. . . .);


. . .  if GradeMade(Grades,100) then . . .
```

Private Type Parameters cont.
and
Restrictions Imposed

What's wrong here?

```
generic
    type Index is (<>);
    type Component is private;
    type Int_Array is array(Index) of Component;
procedure Sort_Array(Arr : in out Int_Array);

procedure Sort_Array(Arr : in out Int_Array) is
  Temp : Component;
begin
  for I in Index'Succ(Arr'First)..Arr'Last loop
    for J in Arr'First..Index'Pred(I) loop
      if Arr(I) < Ar(J) then
        Temp := Arr(J);
        Arr(J) := Arr(I);
        Arr(I) := Temp;
      end if;
    end loop;
  end loop;
end Sort_Array;
```

```
--in user's program

type Student is (Joan,John,Sue,...,Debbie);
type Grade is range 0..100;
type GradeArray is array(Student) of Grade;
function GradeMade is new
   Found(Index=>Student,
         Component=>Grade,
         AnArray=>GradeArray);


Grades : GradeArray := (. . . .);


. . .  if GradeMade(Grades,100) then . . .
```

Private Type Parameters
Another Caution

What's wrong here?

```
generic
  type Element is private;
procedure Swap(X,Y : in out Element);

procedure Swap(X,Y : in out Element) is
   Temp : Element;
begin
  Temp := X;
  X := Y;
  Y := Temp;
end Swap;
```

```
-- in user s program
HerName : string(1..5) := "Lindy";
HisName : string(1..5) := "Chuck";

procedure NameSwap is new Swap(string);
```

???
???

```
procedure NameSwap(X,Y : in out string) is
   Temp : string;  -- OOPS!
begin
  Temp := X;
  X := Y;
  Y := Temp;
end NameSwap;
```

```
generic
  type Element is private;
procedure Swap(X,Y : in out Element);

procedure Swap(X,Y : in out Element) is
   Temp : constant Element :- X;
begin
  X := Y;
  Y := Temp;
end  Swap;
```

==

==

```
procedure NameSwap(X,Y : in out string) is
   Temp : constant string := X;
begin
  X := Y;
  Y := Temp;
end NameSwap;
```

Limited Private Type Parameters

- matches any type including a limited type
- only declaration of objects of the type permitted and NOTHING else

Access Type Parameters

- matches any access type

- operations defined for access types available such as setting object to null, use of NEW allocator, use of .ALL notation

Access Type Parameters
An Example

```
generic
  type Node is private;
  type Link is access Node;
package List is
 . . .
end List;

------------

type STudent;
type StudentPointer is access Student;
type STudent is
  record
    NextStudent, PriorStudent : StudentPointer;
    Name : string(1..20);
    Age    : integer;
  end record;

package StudentPackage is new
    List(Node=>Student, Link=>StudentPointer);
```

Generic Formal Type Parameters
A Synopsis

Generic formal parameter	Actual parameter
type T is limited private;	any type
type T is private;	any non-limited type
type T is (<>);	any discrete type
type T is range<>;	any integer type
type T is digits <>;	any float type
type T is delta <>;	any fixed point type

[*Taken from Ada Language and Methodology by Watt, Wichman, and Findlay]

Type Parameters
and
The Standard Generic IO Packages

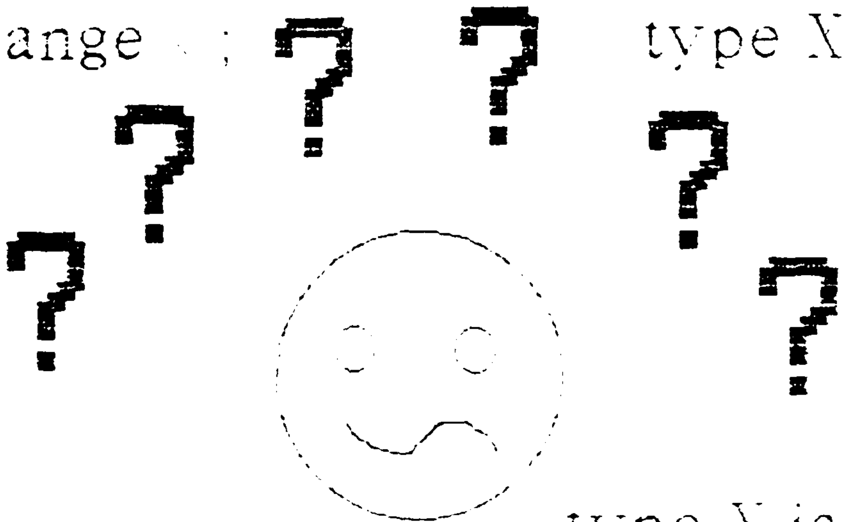
```
package Text_IO is
   . . . non- generic part of Text_IO
   generic
     type NUM is range <>;
   package Integer_IO is
     . . .
   end Integer_IO;

   generic
     type NUM is digits <>;
   package Float_IO is
     . . .
   end Float_IO;

   generic
     type NUM is delta <>;
   package Fixed_IO is
     . . .
   end Fixed_IO;

   generic
     type ENUM is (<>);
   package Enumeration_IO is
     . . .
   end Enumeration_IO;
end Text_IO;
```

How Do I Choose???

type X is digits <>;

type X is range <>;

type X is (<>);



type X is private;

type X is limited private;

Subprogram Parameters
An Example

```
generic
  type Index is (<>);
  type Component is private;
  type Int_Array is array(Index range <>) of
     Component;
  with function "<"(X,Y:Component)
     return boolean;
procedure Sort_Array(Arr : in out Int_Array);

procedure Sort_Array(Arr : in out Int_Array) is
 Temp : Component;
begin
  for I in Index'Succ(Arr'First)..Arr'Last loop
  for J in Arr'First..Index'Pred(I) loop
    if Arr(I) < Ar(J) then
      Temp :- Arr(J);
       Arr(J) := Arr(I);
      Arr(I) :- Temp;
    end if;
  end loop;
  end loop;
end Sort_Array;
```

Generic Formal Type Parameters
How To Choose?

- ☐ What operations are performed on the type in the generic body?
- ☐ How restrictive on the type that the user can choose do you want to be?

Subprogram Parameters

- ☐ allow definition and "pass in" of additional operations for generic formal type parameters - especially private and limited private types

- ☐ can pass functions or procedures

- ☐ formal parameters of generic formal subprogram parameter are checked to ensure match with actual parameters in a call to that subprogram at compile time

Subprogram Parameters

StudentRec

	Age	QPR	Student Number
1	18	3.4	123
	17	2.8	453
	19	1.9	678
	20	2.7	542
	18	3.5	745
	22	3.3	888
	.	.	.
	.	.	.
	.	.	.
	21	3.0	627
	20	2.6	897
30	18	2.2	111

X

Subprogram Parameters - cont.

```
type AnIndex is range 1..100;

type StudentRec is record
  Age : natural;
  QPR : float;
  StudentNumber : natural;
end record;

type StudentArray is array(AnIndex range <>) of StudentRec;

function LT(X,Y : StudentRec) return boolean is
begin
  return X.StudentNumber < Y.StudentNumber;
end LT;

function "<"(X,Y : StudentRec) return boolean is
begin
  return X.QPR < Y.QPR;
end "<";

procedure NumberSort is new Sort_Array
  (Index=>AnIndex, Component=>StudentRec,
   AnArray=>StudentArray, "<" => LT);

procedure QPR_Sort is new Sort_Array
  (Index=>AnIndex, Component=>StudentRec,
   AnArray=>StudentArray, "<"  =>  "<");

StudentData : StudentArray(1..30) := ( .... );
begin
  NumberSort(StudentData);
  QPR_Sort(StudentData);
end;
```

Subprogram Parameters and Default Values

```
generic
  type Index is (<>);
  type Component is private;
  type Int_Array is array(Index range <>) of
     Component;
  with function "<"(X,Y:Component)
     return boolean is <>;
procedure Sort_Array(Arr : in out AnArray);


--in user's program

function "<"(X,Y : StudentRec) return boolean is
begin
  return X.QPR < Y.QPR;
end "<";

procedure DefaultSort is new Sort_Array
  (Index=>AnIndex,Component=>StudentRec,
   AnArray=>StudentArray);


...  DefaultSort(StudentData); -- will sort on
                               -- QPR values
```

Subprogram Parameters and Default Values

```
--in user's program

function LessThan(X,Y : StudentRec) return
    boolean is
begin
  return X.QPR < Y.QPR;
end LessThan;

generic
  type Index is (<>);
  type Component is private;
  type Int_Array is array(Index range <>) of
    Component;
  with function "<"(X,Y:Component)
    return boolean is LessThan;
procedure Sort_Array(Arr : in out AnArray);

procedure DefaultSort is new Sort_Array
  (Index=>AnIndex,Component=>StudentRec,
   AnArray=>StudentArray);


...  DefaultSort(StudentData); -- will sort on
                                -- QPR values
```

Subprogram Parameters and Default Values cont.

Another example:

```
type SmallRange is range 1..10;
type Values is array(SmallRange range <>) of
   integer;

procedure IntegerSort is new Sort_Array
  (Index=>SmallRange, Component=>integer,
   Int_Array=>Values);


  V : Values(5..9) := (. . . .);
begin
  IntegerSort(V);  -- default "<" for integers used
end;

-- using Put for subprogram parameter name
--    results in default to generic Put routines
--    in the IO packages
```

Subprogram Parameters and Subtleties of Default Values

- Global references inside a generic are resolved to those at point of DECLARATION.
- For subprogram parameters, default references resolve to matching names from point of INSTANTIATION.

NAMING CONFUSION

```
with Text_IO; use Text_IO;
procedure Doubles is

   generic
      with procedure [redacted](Char : in character);
      with procedure [redacted](Value: in integer);
   procedure GenericOne;

   procedure GenericOne is
   begin
      DoSomething('A');
      DoSomething(10);
   end GenericOne;

   procedure FirstSomething(Char : in character) is
   begin
      null;
   end FirstSomething;

   procedure SecondSomething(Char : in integer) is
   begin
      null;
   end SecondSomething;

   procedure InstanceOfGenericOne is new
      GenericOne[redacted]>FirstSomething,[redacted]=>SecondSomething);

begin
   InstanceOfGenericOne;
end Doubles;
```

```
with Text_IO; use Text_IO;
package Shell is
  Global : integer := 17;
  generic
    with procedure Put(Val : integer) is <>;
  procedure Demo;
end Shell;

package body Shell is
  procedure Demo is
  begin
    Put(Global);
  end Demo;
end Shell;
----------------------------
with Shell;
package Inner is
  Global : integer := 39;
  procedure Put(I : integer);

  procedure User is new Shell.Demo;
end Inner;

with Text_IO;
package body Inner is
  procedure Put(I : integer) is
  begin
    Text_IO.Put("Surprise" & integer'image(I));
  end Put;
end Inner;

... Inner.User; ...
```

Subprogram Parameters
and
Nesting Generic Units
An Example

```
generic
    type KeyType is private;
    type ElementType is private;
    with function "<"(Left,Right : KeyType)
       return boolean is <>;
package BinaryTreeMaker is
    type Kind is private;
    function Make return Kind;
    function IsEmpty(T : Kind) return boolean;
    procedure Insert(T : in out Kind;
                     K : KeyType;
                     E : ElementType);
    function Retrieve(T : Kind; K : KeyType)
       return ElementType;
    KeyNotFound : exception;

    generic
       with procedure Operation(K : KeyType;
          E : ElementType);
    procedure InorderTraverse(TheTree: in Kind);
private
    type InternalRecord;
    type Kind is access InternalRecord;
end BinaryTreeMaker;
```

(Taken from Understanding Ada by Bray and Pokrass)

```
with EmployeeDataBase; use EmployeeDataBase;
with Text_IO; use Text_IO;
procedure PrintReports is
   package SalaryIO is new Fixed_IO(Dollar);
   package AgeIO is new Integer_IO(AgeType);
   use SalaryIO, AgeIO;

   procedure PrintSalary(Key : NameType;
      Info : EmployeeInfo) is
   begin
     . . . Put(Info.Salary);
   end;

   procedure Print Age(Key : NameType;
      Info : EmployeeInfo) is
   begin
      . . . Put(Info.Age);
   end;

   procedure ReportSalaries is new
      EmployeeTree.InorderTraverse
         (Operation=> PrintSalary);

   procedure ReportAge is new
      EmployeeTree.InorderTraverse
         (Operation=> PrintAge);
begin
   ReportSalaries(RootNode);
   New_Line;
   ReportAges(RootNode);
end PrintReports;
```
(*From Understanding Ada by Bray and Pokrass)

```
with BinaryTreeMaker;
package EmployeeDataBase is
   NameLength : constant := 40;
   subtype NameType is string(1..NameLength);
   type Dollar is delta 0.01 range 0.0..1.0e8;
   type AgeType is range 0 .. 150;
   type YearType is range 1900..2100;
   type EmployeeInfo is record
      Salary : Dollar;
      Age : AgeType;
      Hired : YearType;
   end record;

   package EmployeeTree is new
      BinaryTreeMaker(KeyType=>NameType,
                      ElementType=>EmployeeInfo);

   RootNode : EmployeeTree.Kind;
end EmployeeDataBase;
```

[Taken from Understanding Ada by Bray and Pokrass]

Subprogram Parameters and Handling Exceptions

```
generic
package Stack is
  . . . same as before

  Overflow, Underflow : exception;
end Stack;

-- in user's program

  package S1 is new Stack;
  package S2 is new Stack;

begin
  S1.Push(5);
  S2.Pop(Item);
exception
  when S1.Underflow => . . .;
  when S1.Overflow => . . .;
  when S2.Underflow => . . .;
  when S2.Overflow => . . .;
end;
```

Subprogram Parameters and Handling Exceptions cont.

- Cannot pass exceptions as generic parameter

```
generic
  When_Error : exception;   -- NOT allowed
   . . .
procedure X . . .
  . . .
exception
  when others => raise When_Error;
end X;


My _Exception : exception;
procedure S is new X(My_Exception);

. . .
begin
   S;
exception
  when My_Exception =>  . . .;  -- NOT allowed
end;
```

Subprogram Parameters and Handling Exceptions cont.

```
generic
  with procedure OverflowHandler;
package Stack is
  . . . same as before;
end Stack;

package body Stack is

. . . in Push procedure . . .
  when Constraint_Error => OverflowHandler;

end Stack;

-- in user program
with Stack;
. . .
procedure OverflowHandler is
begin
  Text_IO.Put_Line("Overflow has occurred");
end OverflowHandler;

package S1 is new Stack(OverflowHandler);

begin
  . . .
  S1.Push(5);  -- if overflow occurs msg prints
end;
```

Generic Can'ts

- No generic SUBtype parameters, only TYPEs
- No generic record types
  - No generic tasks
    - Wrap a package around it
- ". . . Ada provides formal types for all classes of type **except record and task** types. The major reason for this is that it is not clear that reasonable criteria for matching exist for these type classes - criteria that would be consistent with the degree of type checking performed elsewhere, yet at the same time have a good probability of being usable for many actual record types and task types." LRM 12.4.2

Tasks within a Generic Package

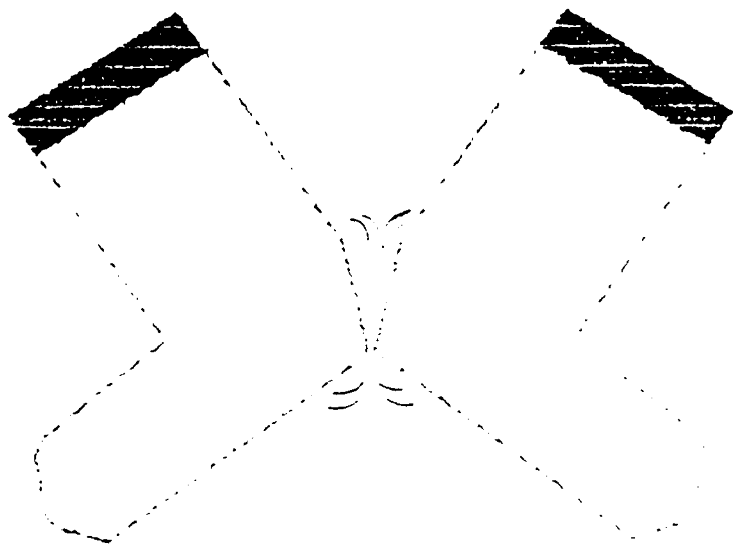
```
generic
  type Item is private;
  Size : Positive := 400;
package On_Buffers is
  task type Buffer is
    entry Read(C : out Item);
    entry Write(C : in Item);
  end;
end On_Buffers;

package body On_Buffers is
  type Length is new Integer range 1.. Size;
  type Vector is array(Length range <>) of Item;
  task body Buffer is
    Pool : Vector(1.. Size);
    Count : Natural := 0;
    In_Index, Out_Index : Length := 1;
  begin
    loop
      select
        when Count < Size =>
          accept Write(C: in Item) do
            Pool(In_Index) := C;
          end;
          In_Index := (In_Index mod Size) + 1;
      or
```

```
        when Count > 0 =>
          accept Read(C : out Item) do
            C := Pool(Out_Index);
          end;
          Out_Index := (Out_Index mod Size) + 1;
          Count := Count - 1;
      or
        terminate;
      end select;
    end loop;
  end Buffer;
end On_Buffers;
```

```
package Character_Buffering is new
   On_Buffers(Item=>character, Size=>100);

A_Buffer : Character_Buffering.Buffer;
```

[Taken from Ada Rationale]

No "Static" Uses



Generic Formal Parameters and Static Uses

- ☐ Generic formal parameters and their attributes NOT allowed constituents of static expressions.

- ☐ No use in case alternatives, type ranges, floating point precisions, etc. (See LRM 4.9)

```
declare
  generic
    X : integer;
  procedure Choice(Val : integer);
  procedure Choice(Val : integer) is
  begin
    case Val is
      when X => . . .          -- illegal usage
      when others => . . .
    end case;
  end Choice;

  procedure TestInstance is new Choice(X=>5);

 begin
   TestInstance(Val=>8);
 end;
```

Generic Formal Parameters and Static Uses (continued)

```
declare
   generic
      X : integer;
   package More_Illegal_Uses is
      type Length is range 1 .. X;
      type Precision is digits X;
      N : constant := X;
   end More_Illegal_Uses;

   package S is new More_Illegal_Uses(3);

begin
   . . .
end;
```

What are the Cons of Generics?

- Takes longer/is harder to write generic code
- Usually some efficiency sacrificed for the generality -- use of application specifics could lead to increased efficiency
- Difficult to make component robust/reliable enough to survive all uses

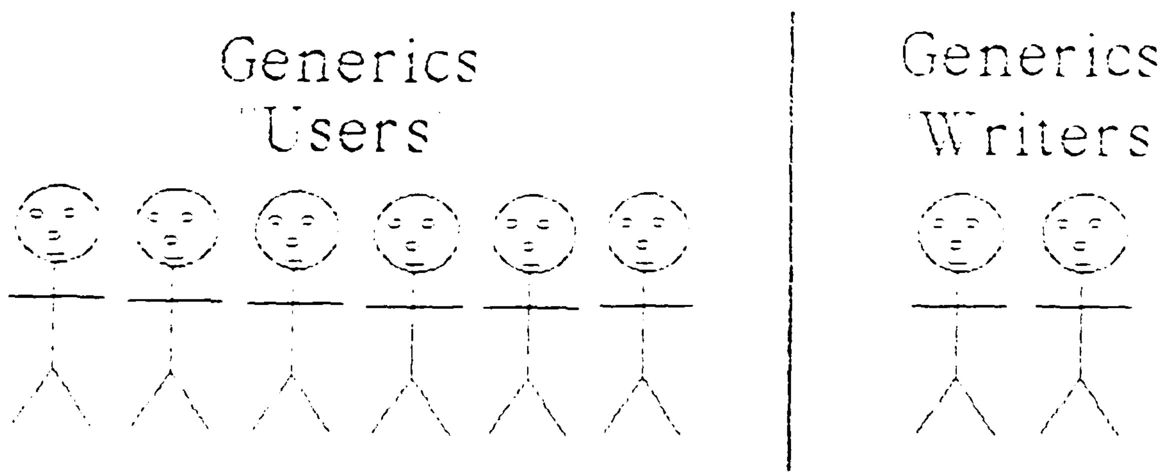
What are the Pros of generics?

- Reusability - no reinventing the wheel for each specific application
- Levels of abstraction added - separation of abstraction and implementation
- Source code size of user programs reduced
  - Maintainability, readability, and understandability increased
  - Verification more manageable
- When used in conjunction with user-defined types increases portability across machines
- Provides necessary answer to strong typing without sacrificing increased reliability of compile time checks
- Provides flexible IO packages which can be used (if needed) for predefined AND user-defined types

Generics Philosophy

(From Ada Rationale)

". . . Whereas such packages are likely to be utilized by LARGE classes of USERS, it should be realized that FEWER programmers will actually be involved in WRITING generic packages. Accordingly we have tried to design a facility that can be almost ignored by the majority of users. They must indeed know how to instantiate a generic package, and this is fairly easy. On the other hand, they need not be familiar with the rules and precautions necessary for writing generic units."



Rationale for Generics

- Construction of general-purpose parameterized packages, procedures and functions
- Units to be used by large classes of users
- Fewer programmers actually involved in writing generic units
- Generic facility can be ignored by majority of Ada users
- Most users only need know how to instantiate a generic unit
- Are context-dependent extension of macro-expansion
- Introduces minimal additional features
- Well implementable within state of art

More on the Generic Model

- Users of generic units should be able to ignore details of generic body entirely

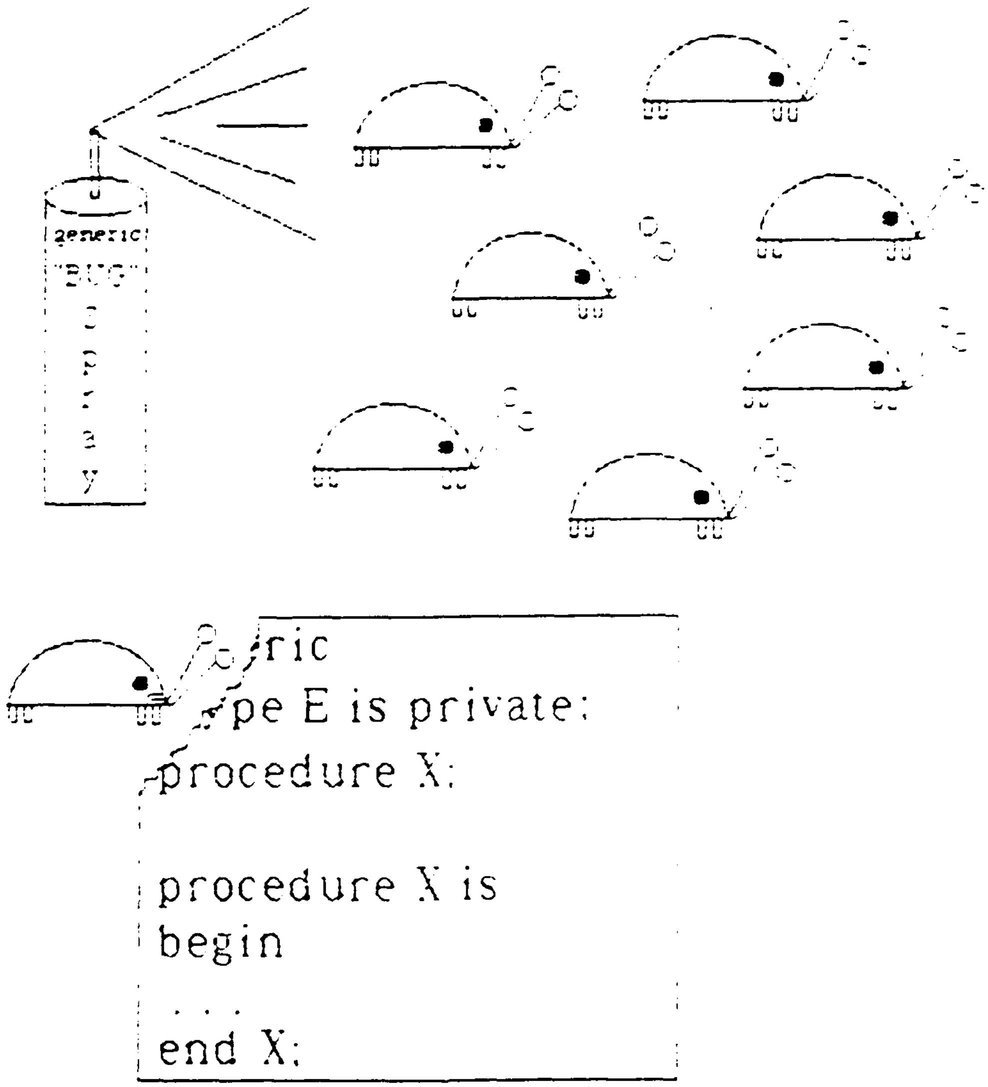
- Errors should be reported to user in terms of the instantiation not body

- Generic body checked for consistency with respect to formal parameter specifications

Unresolved Issues in Generics

- Compiler Issues
  - Use "code sharing" or "code copying" to implement generics
- Management Issues
  - How to facilitate creation of generic units
    - In retrospect, after recognizing similarity in produced units
    - Beforehand using "domain analysis"
  - How to manage storage and retrieval of units in a library of generic units
  - How to "publicize" availability of units in generic library and provide criterion for selecting proper unit
  - How to manage updating of used generic units as "bugs" are uncovered
- Legal Issues
  - Who owns the generic module
  - Who is liable for the generic module's performance

(See Software Components with Ada by Grady Booch)



generic
"BUG"
ric
pe E is private;
procedure X;
procedure X is
begin
. . .
end X;

How do you TEACH generics?

- ☐ Necessary as IO is an issue arising early and should not be kept a "magic" process
- ☐ One key is to use concrete examples
  - ☐ Driver's licence form is a generic template -- individual's license is a usable instantiation
- ☐ One key is to tie to previous learning
  - ☐ Use old/familiar packages, procedures, and functions - Stacks, Swap, etc.

```
with Text_IO, Binary_Search_Trees; use Text_IO;
procedure MidTree is

   type AlphaType is range 1..4000;
   type CompanyType is range 1..36;
   subtype NameType is string(1..20);
   subtype MajorType is string(1..4);

   type MidRec is record
      Alpha   : AlphaType;
      Name    : NameType;
      Company : CompanyType;
      Major   : MajorType;
   end record;

   package AlphaIO is new Integer_IO(AlphaType);
   package CompanyIO is new Integer_IO(CompanyType);
   use AlphaIO, CompanyIO;

   MidFile : File_Type;

   MRec   : MidRec;

   package MidTreePkg is new Binary_Search_Trees(Itemtype=>MidRec);
   use MidTreePkg;

   MidshipmanTree : Tree;

   function "<"(Left,Right : in MidRec) return boolean is
   begin
      return Left.Name < Right.Name;
   end "<";

   procedure Add is new Insert("<"=>"<");

   procedure Print(M : in out MidRec) is
   begin
      Put_Line(M.Name);
   end Print;

   procedure NameList is new LNR_Traversal(Visit=>Print);

begin
   Open(MidFile,In_File,"sys$fac:[moran.play]mids.dat");
   while NOT end_of_file(MidFile) loop
      Get(MidFile,MRec.Alpha);
      Get(MidFile,MRec.Name);
      Get(MidFile,MRec.Company);
      Get(MidFile,MRec.Major);
      Skip_Line(MidFile);
      Add(MidshipmanTree,MRec);
   end loop;
   Close(MidFile);

   NameList(MidshipmanTree);
end;
```

```
with Text_IO, Binary_Search_Trees; use Text_IO;
procedure MoviesTree is

   type CategoryType is (AD, DR, CL, SF, MU, MY);
   subtype IDType is string(1..5);
   subtype LengthType is integer range 0..300;
   subtype YearType is integer range 1800..1988;
   type RatingType is (PG,R,G,NR);
   subtype TitleType is string(1..80);

   type MovieRec is record
      Category : CategoryType;
      ID       : IDType;
      Length   : LengthType;
      Rating   : RatingType;
      Year     : YearType;
      Title    : Titletype;
   end record;

   package IntIO is new Integer_IO(integer);
   package CategoryIO is new Enumeration_IO(CategoryType);
   package RatingIO is new Enumeration_IO(RatingType);
   use IntIO, CategoryIO, RatingIO;

   MovieFile : File_Type;

   MRec   : MovieRec;
   Filler : character;
   Count  : natural;
   Temp   : string(1..80);
   Blanks : string(1..80) := (others=>' ');
   Commando, BearIsland, Daniel, Flashpoint, MassAppeal : MovieRec;

   package MovieTreePkg is new Binary_Search_Trees(Itemtype=>MovieRec);
   use MovieTreePkg;

   MovieTree : Tree;

   function "<"(Left,Right : in MovieRec) return boolean is
   begin
      return Left.Title < Right.Title;
   end "<";

   function EQ(Left,Right : in MovieRec) return boolean is
   begin
      return Left.Title = Right.Title;
   end EQ;

   procedure Add is new InsertByKey("<"=>"<");

   procedure Print(M : in out MovieRec) is
   begin
      Put_Line(M.Title);
   end Print;

   procedure NameList is new LNR_Traversal(Visit=>Print);

   procedure Remove is new RemoveByKey("<"=>"<",EQ=>EQ);

begin
   Commando.Title := Blanks;
```

```
   Daniel.Title(1..6) := "Daniel";
   Flashpoint.Title := Blanks;
   Flashpoint.Title(1..10) := "Flashpoint";
   MassAppeal.Title := Blanks;
   MassAppeal.Title(1..11) := "Mass Appeal";

   Open(MovieFile,In_File,"movies.dat");
   while NOT end_of_file(MovieFile) loop
      Get(MovieFile,MRec.Category);
      Get(MovieFile,Filler);
      Get(MovieFile,MRec.ID);
      Get(MovieFile,Filler);
      Get(MovieFile,MRec.Length);
      Get(MovieFile,Filler);
      Get(MovieFile,MRec.Rating);
      Get(MovieFile,Filler);
      Get(MovieFile,MRec.Year);
      Get(MovieFile,Filler);
      Get_Line(MovieFile,Temp,Count);
      MRec.Title := Blanks;
      MRec.Title(1..Count) := Temp(1..Count);
      Add(MovieTree,MRec);
   end loop;
   Close(MovieFile);

   NameList(MovieTree);
   Remove(MovieTree,BearIsland);
   Remove(MovieTree,Daniel);
   Remove(MovieTree,Flashpoint);
   Remove(MovieTree,MassAppeal);
   Remove(MovieTree,Commando);
   NameList(MovieTree);
end;
```

```
 generic
   type ItemType is private;
package Binary_Search_Trees is

   type Tree is private;

   generic
      with function "<"(Left,Right : in Itemtype) return boolean is <>;
   procedure InsertByKey(T : in out Tree; Item : in Itemtype);

   generic
      with procedure Visit(Item : in out Itemtype);
   procedure NLR_Traversal(T : in Tree);

   generic
      with procedure Visit(Item : in out Itemtype);
   procedure LNR_Traversal(T : in Tree);

   generic
      with procedure Visit(Item : in out Itemtype);
   procedure LRN_Traversal(T : in Tree);

   procedure Share(OriginalTree : in Tree; SharingTree : out Tree);

   procedure Clear(T : out Tree);

   generic
      with function EQ(Left,Right : in Itemtype) return boolean;
      with function "<"(Left,Right : in Itemtype) return boolean;
   procedure RemoveByKey(T : in out Tree; Item : in Itemtype);

   function Left_Son(T : in Tree) return Tree;

   function Right_Son(T : in Tree) return Tree;

   function IsEmpty(T : in Tree) return boolean;

   function GetRootData(T : in Tree) return ItemType;

   Out_Of_Memory : exception;
   Null_Tree     : exception;

private
   type TreeStructure;
   type Tree is access TreeStructure;
end Binary_Search_Trees;

package body Binary_Search_Trees is

   type TreeStructure is record
      Item      : Itemtype;
      LeftSon   : Tree := null;
      RightSon  : Tree := null;
   end record;


   procedure InsertByKey(T : in out Tree; Item : in Itemtype) is
   begin
      if T = null then
         -- found leaf position where Item to be inserted
         -- create new leaf and insert it
```

```
   else
      -- go down right subtree
      InsertByKey(T.RightSon,Item);
   end if;

exception
   when Storage_Error => raise Out_Of_Memory;
end InsertByKey;

procedure NLR_Traversal(T : in Tree) is
begin
   if T /= null then
      Visit(T.Item);
      NLR_Traversal(T.LeftSon);
      NLR_Traversal(T.RightSon);
   end if;
end NLR_Traversal;

procedure LNR_Traversal(T : in Tree) is
begin
   if T /= null then
      LNR_Traversal(T.LeftSon);
      Visit(T.Item);
      LNR_Traversal(T.RightSon);
   end if;
end LNR_Traversal;

procedure LRN_Traversal(T : in Tree) is
begin
   if T /= null then
      LRN_Traversal(T.LeftSon);
      LRN_Traversal(T.RightSon);
      Visit(T.Item);
   end if;
end LRN_Traversal;

procedure Share(OriginalTree : in Tree; SharingTree : out Tree) is
begin
   SharingTree := OriginalTree;
end Share;

procedure Clear(T : out Tree) is
begin
   T := null;
end Clear;

procedure RemoveByKey(T : in out Tree; Item : in ItemType) is
   Father, ReplacementItem : Tree;
 begin
   if T = null then
      -- do nothing...item not in the tree
      null;
   elsif EQ(Item, T.Item) then
      if (T.RightSon=null) and (T.LeftSon=null) then
         -- item is a leaf...no reattachment of children necessary
         T := null;
      else   -- item not a leaf
         -- go left and then right as far as possible to find
         -- replacement "value" to put in deleted place
         if T.LeftSon /= null then
            Father := T;
            ReplacementItem := T.LeftSon;
```

```
              -- transfer replacement value up into position
              T.Item := ReplacementItem.Item;
              -- reattach children of replacement value that
              -- was pulled up
              if Father = T then
                 T.LeftSon := ReplacementItem.LeftSon;
              else
                 Father.RightSon := ReplacementItem.LeftSon;
              end if;
           else
              -- go right and then left as far as possible to find
              -- replacement "value" to put in deleted place
              Father := T;
              ReplacementItem := T.RightSon;
              while ReplacementItem.LeftSon /= null loop
                 Father := ReplacementItem;
                 ReplacementItem := ReplacementItem.LeftSon;
              end loop;
              -- transfer replacement value up into position
              T.Item := ReplacementItem.Item;
              -- reattach children of replacement value that
              -- was pulled up
              if Father = T then
                 T.RightSon := ReplacementItem.RightSon;
              else
                 Father.LeftSon := ReplacementItem.RightSon;
              end if;
           end if;
        end if;
     elsif Item < T.Item then
        -- go down left subtree
        RemoveByKey(T.LeftSon,Item);
     else
        -- go down right subtree
        RemoveByKey(T.RightSon,Item);
     end if;
  end RemoveByKey;

  function Left_Son(T : in Tree) return Tree is
  begin
     if T = null then
         raise Null_Tree;
     else
         return T.LeftSon;
     end if;
  end Left_Son;

  function Right_Son(T : in Tree) return Tree is
  begin
     if T = null then
        raise Null_Tree;
     else
        return T.RightSon;
     end if;
  end Right_Son;

  function IsEmpty(T : in Tree) return boolean is
  begin
     return T = null;
  end IsEmpty;

  function GetRootData(T : in Tree) return ItemType is
```

```
            return T.Item;
          end if;
        end GetRootData;

      end Binary_Search_Trees;
```

```
with Lists, Text_IO; use Text_IO;
procedure MoviesList is

   type CategoryType is (AD, DR, CL, SF, MU, MY);
   subtype IDType is string(1..5);
   subtype LengthType is integer range 0..300;
   subtype YearType is integer range 1800..1988;
   type RatingType is (PG,R,G,NR);
   subtype TitleType is string(1..80);

   type MovieRec is record
      Category : CategoryType;
      ID       : IDType;
      Length   : LengthType;
      Rating   : RatingType;
      Year     : YearType;
      Title    : Titletype;
   end record;

   package IntIO is new Integer_IO(integer);
   package CategoryIO is new Enumeration_IO(CategoryType);
   package RatingIO is new Enumeration_IO(RatingType);
   use IntIO, CategoryIO, RatingIO;

   MovieFile : File_Type;

   MRec   : MovieRec;
   Filler : character;
   Count  : natural;
   Temp   : string(1..80);
   Blanks : string(1..80) := (others=>' ');

   function Get_Title(Movie : MovieRec) return Titletype;
   function "<"(Left, Right : TitleType) return boolean;
   function EQ(Left, Right : TitleType) return boolean;

   package MovieListPkg is new Lists(Item=>MovieRec,
                                     KeyType=>TitleType,
                                     Key=>Get_Title,
                                     LE=>"<", EQ=>EQ);
   use MovieListPkg;

   MovieList : ListPointer;

   function Get_Title(Movie : MovieRec) return TitleType is
   begin
      return Movie.Title;
   end Get_Title;

   function "<"(Left,Right : TitleType) return boolean is
   begin
      return Left < Right;
   end "<";

   function EQ(Left,Right : TitleType) return boolean is
   begin
      return Left = Right;
   end EQ;

begin
   Open(MovieFile,In_File,"movies.dat");
```

```
        Get(MovieFile,Filler);
        Get(MovieFile,MRec.Length);
        Get(MovieFile,Filler);
        Get(MovieFile,MRec.Rating);
        Get(MovieFile,Filler);
        Get(MovieFile,MRec.Year);
        Get(MovieFile,Filler);
        Get_Line(MovieFile,Temp,Count);
        MRec.Title := Blanks;
        MRec.Title(1..Count) := Temp(1..Count);
        Put(MRec.Title(1..Count));
        InsertInOrderInList(MovieList,MRec);
     end loop;
     Close(MovieFile);

  end;
```

```
-- Module        : Lists
-- Author        : LCDR MORAN
-- Date          : 29 SEP 1987
-- Function      : Implements basic operations on a singly linked list.

generic
   type Item is private;
   type KeyType is private;

   with function Key(AnItem : Item) return KeyType;
   with function LE(Key1, Key2 : KeyType) return boolean;
   with function EQ(Key1, Key2 : KeyType) return boolean;
package Lists is

   subtype Count is natural;

   type ListPointer is private;

   procedure Copy(PointerToOriginalList : in ListPointer;
                  PointerToCopyList     : out ListPointer);

   procedure Clear(PointerToTheList : in out ListPointer);

   procedure Share(PointerToOriginalList,
                   PointerToSharingList : in out ListPointer);

   procedure InsertAtHeadOfList(PointerToTheList      : in out ListPointer;
                                TheItemToBeInserted   : in Item);

   procedure InsertAtTailOfList(PointerToTheList      : in out ListPointer;
                                TheItemToBeInserted   : in Item);

   procedure InsertInOrderInList(PointerToTheList     : in out ListPointer;
                                 TheItemToBeInserted  : in Item);

   procedure RemoveFromHeadOfList(PointerToTheList    : in out ListPointer;
                                  RemovedItem         : out Item);

   procedure RemoveFromTailOfList(PointerToTheList    : in out ListPointer;
                                  RemovedItem         : out Item);

   procedure RemoveByKeyFromList(PointerToTheList     : in out ListPointer;
                                 RemovedItem          : out Item;
                                 KeyValue             : in KeyType);

   function AreEqual(PointerToL1, PointerToL2 : ListPointer) return boolean

   function IsEmpty(PointerToL : ListPointer) return boolean;

   function LengthOf(PointerToL : ListPointer) return Count;

   function Predecessor(PointerToAList, PointerToANode : ListPointer)
                     return ListPointer;

   function Successor(PointerToAList, PointerToANode : ListPointer)
                     return ListPointer;

   function GetData(PointerToANode : ListPointer) return Item;

   EmptyList : exception;
```

```
private
   type ListNode;
   type ListPointer is access ListNode;

end Lists;
```

```
-- Module        : Lists
-- Author        : LCDR MORAN
-- Date          : 29 SEP 1987
-- Function      : Implements basic operations on a singly linked list.
with Unchecked_Deallocation;
package body Lists is

   type ListNode is record
       Data : Item;
       NextPointer : ListPointer;
   end record;

   function Successor(PointerToAList, PointerToANode : ListPointer)
                         return ListPointer is
   begin
      return PointerToANode.NextPointer;
   end Successor;

   function Predecessor(PointerToAList, PointerToANode : ListPointer)
                          return ListPointer is
      Prior, Temp : ListPointer := PointerToAList;
   begin
      if PointerToANode = PointerToAList then
         return null;
      else
         while Temp /= null and Temp /= PointerToANode loop
            Prior := Temp;
            Temp := Temp.NextPointer;
         end loop;
         if Temp /= null then
            return Prior;
         else
            return null;
         end if;
      end if;
   end Predecessor;

   function GetData(PointerToANode : ListPointer) return Item is
   begin
      if PointerToANode /= null then
         return PointerToANode.Data;
      end if;
   end GetData;

   procedure Dispose is new Unchecked_Deallocation(ListNode,ListPointer);

   procedure Copy(PointerToOriginalList : in ListPointer;
                  PointerToCopyList     : out ListPointer) is
      Temp : ListPointer := PointerToOriginalList;
      LastAddedPtr : ListPointer;
      NewNodePtr : ListPointer;
   begin
      PointerToCopyList := null;
      while Temp /= null loop
         -- make the new node and copy the data into it
         NewNodePtr := new ListNode;
         NewNodePtr.Data := Temp.Data;

         if Temp = PointerToOriginalList then   -- add the first node
            PointerToCopyList := NewNodePtr;
```

```
         else                                      -- add other than the first no
            LastAddedPtr.NextPointer := NewNodePtr;
         end if;

         Temp := Temp.NextPointer;            -- move to next node in orig. li
         LastAddedPtr := NewNodePtr;          -- keep track of last node added
      end loop;
   end Copy;

   procedure Clear(PointerToTheList : in out ListPointer) is
      Temp, Trail : ListPointer := PointerToTheList;
   begin
      while Temp /= null loop
         Trail := Temp;
         Temp := Temp.NextPointer;
         Dispose(Trail);
      end loop;

      PointerToTheList := null;
   end Clear;

   procedure Share(PointerToOriginalList,
                   PointerToSharingList : in out ListPointer) is
   begin
      PointerToSharingList := PointerToOriginalList;
   end Share;

   function IsEmpty(PointerToL : ListPointer) return boolean is
   begin
      return (PointerToL = null);
   end IsEmpty;

   procedure InsertAtHeadOfList(PointerToTheList    : in out ListPointer;
                                TheItemToBeInserted : in Item) is
      PointerToNewNodeToBeInserted : ListPointer;
   begin
      PointerToNewNodeToBeInserted := new ListNode;
      PointerToNewNodeToBeInserted.Data := TheItemToBeInserted;
      if NOT IsEmpty(PointerToTheList) then
         PointerToNewNodeToBeInserted.NextPointer := PointerToTheList;
      end if;
      PointerToTheList := PointerToNewNodeToBeInserted;
   end InsertAtHeadOfList;

   procedure InsertAtTailOfList(PointerToTheList    : in out ListPointer;
                                TheItemToBeInserted : in Item) is
      TempPointer : ListPointer;
      PointerToNewNodeToBeInserted : ListPointer;
   begin
      PointerToNewNodeToBeInserted := new ListNode;
      PointerToNewNodeToBeInserted.Data := TheItemToBeInserted;
      if IsEmpty(PointerToTheList) then
         InsertAtHeadOfList(PointerToTheList,TheItemToBeInserted);
      else
         TempPointer := PointerToTheList;
         while TempPointer.NextPointer /= null
         loop
            TempPointer := TempPointer.NextPointer;
         end loop;
         TempPointer.NextPointer := PointerToNewNodeToBeInserted;
```

```
   end if;
end InsertAtTailOfList;

procedure InsertInOrderInList(PointerToTheList    : in out ListPointer;
                              TheItemToBeInserted : in Item) is
   Temp, Trail : ListPointer := PointerToTheList;
   PointerToTheNewNodeToBeInserted : ListPointer;
begin
   if IsEmpty(PointerToTheList) or else
      (NOT LE(Key(PointerToTheList.Data),Key(TheItemToBeInserted))) then
      InsertAtHeadOfList(PointerToTheList,TheItemToBeInserted);
   else
      while (Temp /= null) and then
            (LE(Key(Temp.Data),Key(TheItemToBeInserted))) loop
         Trail := Temp;
         Temp := Temp.NextPointer;
      end loop;
      PointerToTheNewNodeToBeInserted := new ListNode;
      PointerToTheNewNodeToBeInserted.Data := TheItemToBeInserted;
      Trail.NextPointer := PointerToTheNewNodeToBeInserted;
      PointerToTheNewNodeToBeInserted.NextPointer := Temp;
   end if;
end InsertInOrderInList;

procedure RemoveFromHeadOfList(PointerToTheList  : in out ListPointer;
                               RemovedItem       : out Item) is
   Temp : ListPointer := PointerToTheList;
begin
   if IsEmpty(PointerToTheList) then
      raise EmptyList;
   else
      RemovedItem := PointerToTheList.Data;
      PointerToTheList := PointerToTheList.NextPointer;
      Dispose(Temp);
   end if;
end RemoveFromHeadOfList;

procedure RemoveFromTailOfList(PointerToTheList  : in out ListPointer;
                               RemovedItem       : out Item) is
   TempPointer, PriorPointer : ListPointer;
begin
   if IsEmpty(PointerToTheList) then
      raise EmptyList;
   elsif PointerToTheList.NextPointer = null then
      RemoveFromHeadOfList(PointerToTheList, RemovedItem);
   else
      TempPointer := PointerToTheList;
      while TempPointer.NextPointer /= null
      loop
         PriorPointer := TempPointer;
         TempPointer := TempPointer.NextPointer;
      end loop;
      RemovedItem := TempPointer.Data;
      Dispose(TempPointer);
      PriorPointer.NextPointer := null;
   end if;
end RemoveFromTailOfList;

procedure RemoveByKeyFromList(PointerToTheList : in out ListPointer;
                              RemovedItem      : out Item;
```

```
                                        KeyValue        : in Keytype) is
      TempPointer, PriorPointer : ListPointer;
   begin
      if IsEmpty(PointerToTheList) then
         raise EmptyList;
      elsif EQ(Key(PointerToTheList.Data),KeyValue) then
         RemoveFromHeadOfList(PointerToTheList, RemovedItem);
      else
         TempPointer := PointerToTheList;
         while (TempPointer /= null) and then
               (NOT EQ(Key(TempPointer.Data),KeyValue))
         loop
            PriorPointer := TempPointer;
            TempPointer := TempPointer.NextPointer;
         end loop;
         if TempPointer /= null then
            RemovedItem := TempPointer.Data;
            PriorPointer.NextPointer := TempPointer.NextPointer;
            Dispose(TempPointer);
         else
            raise EmptyList;
         end if;
      end if;
   end RemoveByKeyFromList;

   function AreEqual(PointerToL1, PointerToL2 : ListPointer) return boolean is
      TempPointerToL1 : ListPointer := PointerToL1;
      TempPointerToL2 : ListPointer := PointerToL2;
   begin
      while (TempPointerToL1.Data = TempPointerToL2.Data) and
            (TempPointerToL1 /= null) and (TempPointerToL2 /= null)
      loop
         TempPointerToL1 := TempPointerToL1.NextPointer;
         TempPointerToL2 := TempPointerToL2.NextPointer;
      end loop;
      if (TempPointerToL1 = null) and (TempPointerToL2 = null) then
         return true;
      elsif (TempPointerToL1 = null) and (TempPointerToL2 /= null) then
         return false;
      elsif (TempPointerToL1 /= null) and (TempPointerToL2 = null) then
         return false;
      else
         return (TempPointerToL1.Data = TempPointerToL2.Data);
      end if;
   end AreEqual;

   function LengthOf(PointerToL : ListPointer) return Count is
      TempPointer : ListPointer := PointerToL;
      Length : Count := 0;
   begin
      while TempPointer /= null
      loop
         Length := Length + 1;
         TempPointer := TempPointer.NextPointer;
      end loop;
      return Length;
   end LengthOf;

end Lists;
```

```
with Lists;
package Polynomials is

   subtype CoefficientType is integer;
   subtype ExponentType is integer;

   type Term is record
      Coefficient : CoefficientType;
      Exponent    : ExponentType;
   end record;

   function ExponentValue(ATerm : Term) return ExponentType;

   function LE(Exponent1, Exponent2 : ExponentType) return boolean;

   function EQ(Exponent1, Exponent2 : ExponentType) return boolean;

   package PolynomialLists is new Lists(Item=>Term,KeyType=>ExponentType,
                                        LE => LE, EQ => EQ,
                                        Key => ExponentValue);
   use PolynomialLists;

   subtype Polynomial is ListPointer;

   function CreatePolynomial(InputFile : string) return Polynomial;

   function "+"(P1,P2 : Polynomial) return Polynomial;

   procedure Put(P : in Polynomial);

end Polynomials;
```

```
with Text_IO; use Text_IO;
package body Polynomials is

   function NoMoreTerms(P : Polynomial) return boolean renames
      PolynomialLists.IsEmpty;

   function TermValue(P : Polynomial) return Term renames
      PolynomialLists.GetData;

   procedure AddTermToPolynomial(P : in out Polynomial; ATerm : in Term)
      renames PolynomialLists.InsertInOrderInList;

   function MoreTerms(P : Polynomial) return boolean is
   begin
      return NOT (NoMoreTerms(P));
   end MoreTerms;

   function ExponentValue(ATerm : Term) return ExponentType is
   begin
      return ATerm.Exponent;
   end ExponentValue;

   function CoefficientValue(ATerm : Term) return CoefficientType is
   begin
      return ATerm.Coefficient;
   end CoefficientValue;

   function LE(Exponent1, Exponent2 : ExponentType) return boolean is
   begin
      return Exponent1 <= Exponent2;
   end LE;

   function EQ(Exponent1, Exponent2 : ExponentType) return boolean is
   begin
      return Exponent1 = Exponent2;
   end EQ;

   function CreatePolynomial(InputFile : string) return Polynomial is
      ATerm : Term;
      PolynomialFile : file_type;
      P : Polynomial;
      package Int_IO is new Integer_IO(integer);
      use Int_IO;
   begin
      Open(PolynomialFile,In_File,InputFile);
      while NOT end_of_file(PolynomialFile)
      loop
         Get(PolynomialFile,ATerm.Coefficient);
         Get(PolynomialFile,ATerm.Exponent);
         if ATerm.Coefficient /= 0 then
            AddTermToPolynomial(P,ATerm);
         end if;
      end loop;
      return P;
   exception
      when Name_Error=>Put_Line("ERROR - Nonexistent file");
      when Data_Error=>Put_Line("ERROR - Data error in file");
   end CreatePolynomial;

   function "+"(P1,P2 : Polynomial) return Polynomial is
```

```
   Temp1 : Polynomial := P1;
   Temp2 : Polynomial := P2;
   Sum   : Polynomial;
   Tail  : Polynomial;
begin
   if IsEmpty(P1) then
      Copy(P2,Sum);
   elsif IsEmpty(P2) then
      Copy(P1,Sum);
   else
      while (MoreTerms(Temp1) and MoreTerms(Temp2))
      loop
         while (MoreTerms(Temp1) and MoreTerms(Temp2)) and then
           (ExponentValue(TermValue(Temp1))=ExponentValue(TermValue(Temp2)
         loop
            if (CoefficientValue(TermValue(Temp1)) +
                CoefficientValue(TermValue(Temp2))) /= 0 then
               AddTermToPolynomial(Sum,(CoefficientValue(TermValue(Temp1)
                                       +CoefficientValue(TermValue(Temp2))
                                       ExponentValue(TermValue(Temp1))));
            end if;
            Temp1 := Successor(P1,Temp1);
            Temp2 := Successor(P2,Temp2);
         end loop;

         while (MoreTerms(Temp1) and MoreTerms(Temp2)) and then
           (ExponentValue(TermValue(Temp1))<ExponentValue(TermValue(Temp2)
         loop
            AddTermToPolynomial(Sum,(CoefficientValue(TermValue(Temp1)),
                                    ExponentValue(TermValue(Temp1))));
            Temp1 := Successor(P1,Temp1);
         end loop;

         while (MoreTerms(Temp1) and MoreTerms(Temp2)) and then
           (ExponentValue(TermValue(Temp2))<ExponentValue(TermValue(Temp1)
         loop
            AddTermToPolynomial(Sum,(CoefficientValue(TermValue(Temp2)),
                                    ExponentValue(TermValue(Temp2))));
            Temp2 := Successor(P2,Temp2);
         end loop;
      end loop;
   end if;

   if MoreTerms(Temp2) then
      Temp1 := Temp2;
   end if;
   while MoreTerms(Temp1) loop
      AddTermToPolynomial(Sum,(CoefficientValue(TermValue(Temp1)),
                              ExponentValue(TermValue(Temp1))));
      Temp1 := Successor(P1,Temp1);
   end loop;

   return Sum;
end "+";

procedure Put(P : in Polynomial) is
   Temp : Polynomial := P;
   package Int_IO is new Integer_IO(integer);
   use Int_IO;
begin
```

```
      while MoreTerms(Temp)  loop
         if CoefficientValue(TermValue(Temp)) > 0 then
            Put('+');
         end if;
         Put(CoefficientValue(TermValue(Temp)),0);
         Put("X^");
         Put(ExponentValue(TermValue(Temp)),0);
         Temp := Successor(P,Temp);
      end loop;
   end Put;

end Polynomials;
```

```
with Polynomials, Text_IO; use Polynomials, Text_IO;
procedure AddPolynomials is

   FirstPolynomial,SecondPolynomial : string(1..30) :=
      "                              ";

   procedure GetPolynomialFileName(FileName : out string) is
     NumChars : natural;
     TFileName : string(1..30);
   begin
      New_Line(2);
      Put_Line("Enter filename where a polynomial is located.");
      Get_Line(TFileName,NumChars);
      FileName := TFilename(1..NumChars);
   end GetPolynomialFileName;

begin
   New_Page;
   GetPolynomialFileName(FirstPolynomial);
   GetPolynomialFileName(SecondPolynomial);
   Put(CreatePolynomial(FirstPolynomial) + CreatePolynomial(SecondPolynomial
end AddPolynomials;
```

```
generic
    type Item  is private;
    type Index is (<>);
    type Items is array(Index range <>) of Item;
    with function "<" (Left  : in Item;
                       Right : in Item) return Boolean;
package Heap_Sort is

    procedure Sort (The_Items : in out Items);

end Heap_Sort;
```

```
generic
    type Item  is private;
    type Index is (<>);
    type Items is array(Index range <>) of Item;
    with function "<" (Left  : in Item;
                       Right : in Item) return Boolean;
package Quick_Sort is

    procedure Sort (The_Items : in out Items);

end Quick_Sort;
```

```
generic
    type Item  is private;
    type Index is (<>);
    type Items is array(Index range <>) of Item;
    with function "<" (Left  : in Item;
                       Right : in Item) return Boolean;
package Binary_Insertion_Sort is

    procedure Sort (The_Items : in out Items);

end Binary_Insertion_Sort;
```

[Taken from Software Components with Ada by Grady Booch]

```
generic
    type Key   is limited private;
    type Item  is limited private;
    type Index is (<>);
    type Items is array(Index range <>) of Item;
    with function Is_Equal (Left  : in Key;
                            Right : in Item) return Boolean;
package Sequential_Search is

    function Location_Of (The_Key      : in Key;
                          In_The_Items : in Items) return Index;

    Item_Not_Found : exception;

end Sequential_Search;
```

```
generic
    type Key   is limited private;
    type Item  is limited private;
    type Index is (<>);
    type Items is array(Index range <>) of Item;
    with function Is_Equal     (Left  : in Key;
                                Right : in Item) return Boolean;
    with function Is_Less_Than (Left  : in Key;
                                Right : in Item) return Boolean;
package Ordered_Sequential_Search is

    function Location_Of (The_Key      : in Key;
                          In_The_Items : in Items) return Index;

    Item_Not_Found : exception;

end Ordered_Sequential_Search;
```

```
generic
    type Key   is limited private;
    type Item  is limited private;
    type Index is (<>);
    type Items is array(Index range <>) of Item;
    with function Is_Equal (Left  : in Key;
                            Right : in Item) return Boolean;
    with function Is_Less_Than (Left  : in Key;
                                Right : in Item) return Boolean;
package Binary_Search is

    function Location_Of (The_Key      : in Key;
                          In_The_Items : in Items) return Index;

    Item_Not_Found : exception;

end Binary_Search;
```

[Taken from Software Components with Ada by Grady Booch]

David A. Cook
INSTRUCTOR COMPUTER SCIENCE
U S AIR FORCE ACADEMY CO 80840

472-3590
AV 259-3590
DFCS

HOME 472-6935
QTRS 4206F
USAFA CO 8084(

Ada* Tasking
Abstraction of Process

Captain David A. Cook

U.S. Air Force Academy

*Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

ADA TASKING

- OVERVIEW

  DEFINE ADA TASKING

  DEFINE SYNCHRONIZATION MECHANISM

  EXAMPLES

ADA TASKING

TASK DEFINITION

- A PROGRAM UNIT FOR CONCURRENT EXECUTION
- NEVER A LIBRARY UNIT
- MASTER IS A . . .
  - LIBRARY PACKAGE
  - SUBPROGRAM
  - BLOCK STATEMENT
  - OTHER TASK

ADA TASKING

SYNCHRONIZATION MECHANISMS

- GLOBAL VARIABLES
- RENDEZVOUS

  MAIN PROGRAM I~~N~~S A TASK

  CALLER REQUESTS SERVICE

  1. IMMEDIATE REQUEST
  2. WAIT FOR A WHILE
  3. WAIT FOREVER

CALLEE PROVIDES SERVICE

1. IMMEDIATE RESPONSE
2. WAIT FOR A WHILE
3. WAIT FOREVER

SERVICE IS REQUESTED WITH AN ENTRY CALL STATEMENT

SERVICE IS PROVIDED WITH AN ACCEPT STATEMENT

ADA TASKING

SELECT STATEMENTS PROVIDE ABILITY TO PROGRAM THE DIFFERENT REQUEST AND PROVIDE MODES

GUARDS ARE "IF STATEMENTS" FOR PROVIDING SERVICE [True or False condition]

TERMINATION IS AN ALTERNATIVE IF A SERVICE IS NO LONGER NEEDED

TASK MASTERS

EACH TASK MUST DEPEND ON A MASTER

A MASTER CAN BE A TASK, A CURRENTLY EXECUTING BLOCK STATEMENT, A CURRENTLY EXECUTING SUBPROGRAM, OR A LIBRARY PACKAGE.

PACKAGES DECLARED INSIDE ANOTHER PROGRAM UNIT CANNOT BE MASTERS.

THE MASTER OF A TASK IS DETERMINED BY THE CREATION OF THE TASK OBJECT.

A BLOCK, TASK, OR SUBPROGRAM CANNOT BE LEFT UNTIL ALL OF ITS DEPENDENTS ARE TERMINATED.

- FOR THE MAIN PROGRAM, TERMINATION DOES NOT DEPEND ON TASK WHOSE MASTER IS A LIBRARY PACKAGE.

- ACTUALLY, THE 1815A DOES NOT DEFINE IF TASKS THAT DEPEND ON LIBRARY PACKAGES ARE REQUIRED TO TERMINATE!!

WHEN DOES A TASK START?

TASKS ARE ACTIVATED AFTER THE ELABORATION OF THE DECLARATIVE PART.

EFFECTIVELY, ACTIVATION IS AFTER THE DECLARATIVE PART, AND IMMEDIATELY AFTER THE 'BEGIN' STATEMENT, BUT BEFORE ANY OTHER STATEMENT.

THE PURPOSE OF THIS IS TO ALLOW THE EXCEPTION HANDLER TO SERVICE TASKING EXCEPTION.

```
Task type T1 is ....
Obj : T1;

begin
    declare
    New_Obj:T1;

    begin
       null;
    end;

...
...
end;
```

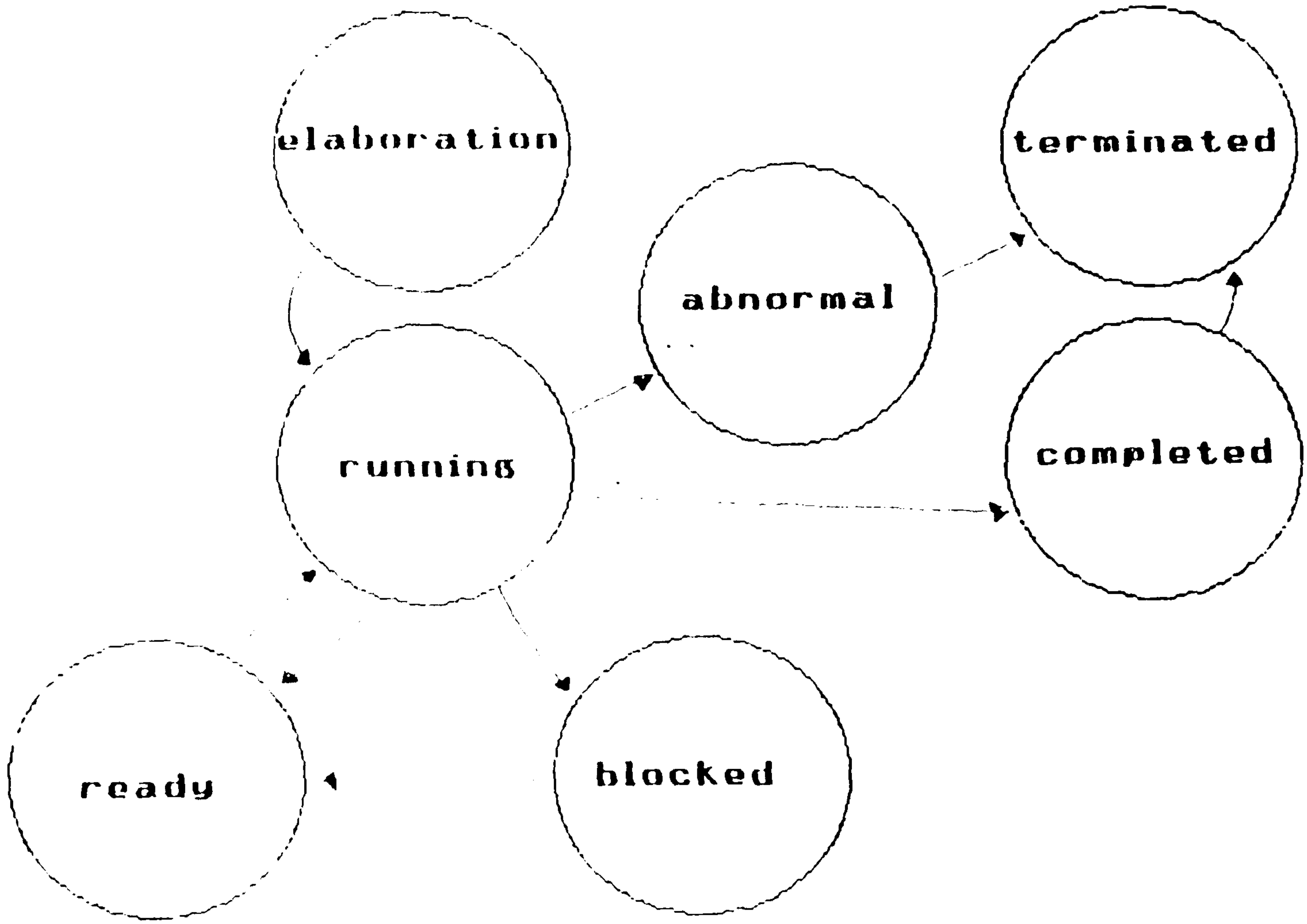
TASKS OBJECTS ACCESSED BY ALLOCATORS DO THINGS A LITTLE BIT DIFFERENTLY

NORMALLY, THE SCOPE OF A TASK OBJECT DETERMINES ITS MASTER

FOR AN ACCESS TYPE, THE MASTER IS DETERMINED BY THE ACCESS TYPE DEFINITION

ACTIVATION FOR ACCESSED TASKS OCCURS IMMEDIATELY UPON THE ASSIGNMENT OF A VALUE TO THE ACCESS OBJECT

```
Task Type T1 is...
Obj : T1;
Type T1_Ptr is access T1;
Ptr_Obj : T1_Ptr := new T1;

begin
    declare
    New_Ptr_Obj:T1_Ptr:=new T1;

    begin
       null;
    end;
...
...
end;
```


elaboration
terminated
abnormal
running
completed
ready
blocked

ELABORATIUN - UECLARATIVE PART

RUNNING - TASK HAS PROCESSOR

READY - TASK IS AVAILABLE FOR PROCESSOR, AND HAS ALL RESOURCES TO RUN

BLOCKED - TASK IS EITHER WAITING FOR A CALL, OR WAITING FOR CALL TO BE ANSWERED

COMPLETED - AT END, OR EXCEPTION

TERMINATED - COMPLETED, AND DEPENDENT TASKS ALSO TERMINATED

ABNORMAL - TASK WAS ABORTED

```
task [type] [is
   {entry_declaration}
   {representation_clause}
end [task_simple_name] ]


task body task_simple_name is
   [declarative_part]
begin
   [sequence_of_statements]
[exception
   exception_handler
  {exception_handler}]
end [task_simple_name];
```

ACCEPT STATEMENT

THE ACCEPT STATEMENT ALLOWS AN UNKNOWN CALLER TO CALL AN ENTRY.

THERE CAN BE IN AND/OR OUT PARAMETERS

THE CONSTRUCT IS 'ACCEPT.....DO'

DURING THE ACCEPT, THE CALLING UNIT IS SUSPENDED. THUS, A LONG ACCEPT SLOWS DOWN THE SYSTEM.

A GOOD APPROACH IS TO USE THE ACCEPT SIMPLY TO COPY IN OR OUT DATA, AND ALLOW THE CALLER TO CONTINUE.

SIMPLEST FORM OF TASK ENTRY

ACCEPT

```
TASK T1 IS
    ENTRY ENTRY1;
END T1;
.
.
TASK BODY T1 IS
  BEGIN
    LOOP

          ACCEPT ENTRY1 DO
                <SOS>
          END ENTRY1;
     <SOS>

     END LOOP;
  END T1;
--WAIT FOREVER FOR CALL TO ENTRY1
```

```
TASK T1 IS
  ENTRY ACTION (DATA : SOME_TYPE);
END T1;

TASK BODY T1 IS

  BEGIN
    LOOP
      ACCEPT ACTION(DATA:SOME_TYPE) DO
        --SOME LONG PROCESS USING DATA
        --  OCCURS HERE
      END ACTION;
    END LOOP;
  END T1;

--NO EXITS OR GOTOS ALLOWED IN ACCEPT,
--  BUT A RETURN IS ALLOWED
```

```
TASK T1 IS
  ENTRY ACTION (DATA : SOME_TYPE);
END T1;


TASK BODY T1 IS
  LOCAL : SOME_TYPE;
  BEGIN
    LOOP
      ACCEPT ACTION(DATA:SOME_TYPE) DO
        LOCAL := DATA;
      END ACTION;
       --PUT PROCESS ON LOCAL HERE
    END LOOP;
  END T1;
--WHEN THIS CAN BE DONE, IT WILL SPEED
--UP THE SYSTEM.
```

```
TASK T1 IS
  ENTRY ACTION(DATA:A_TYPE);
  ENTRY RESULT(DATA :OUT A_TYPE);
END T1;

TASK BODY T1 IS
  LOCAL : A_TYPE;
  BEGIN
    LOOP
      ACCEPT ACTION(DATA:A_TYPE) DO
        LOCAL := DATA;
      END ACTION;
      --PROCESS ON LOCAL HERE
      ACCEPT R[illegible]T(DATA:OUT A_TYPE) DO
        DATA :[illegible]
      END RES[illegible]
    END LOOP;
  END T1;
```

```
TASK T1 IS
     ENTRY ENTRY1;
END T1;
.
.
TASK BODY T1 IS
  BEGIN
    LOOP

              ACCEPT ENTRY1; --'SYNC' CALL ONLY
         <SOS>

         END LOOP;
    END T1;
--WAIT FOREVER FOR CALL TO ENTRY1


--EVEN IF ENTRY1 HAS PARAMETERS ASSOCIATED WITH
--     IT, THE ACCEPT BLOCK DOES NOT HAVE TO HAVE A
--     SEQUENCE OF STATEMENTS
```

SELECT STATEMENT

USED BY THE TASK TO ALLOW OPTIONS

SIMPLEST FORM IS THE SELECTIVE WAIT (WAIT FOREVER)

```
TASK T1 IS
    ENTRY ENTRY1;
    ENTRY ENTRY2;
END T1;
.
.
.
TASK BODY T1 IS
 BEGIN
   LOOP
      SELECT
         ACCEPT ENTRY1 DO
           <SOS>
         END ENTRY1;
         <SOS>
      OR
         ACCEPT ENTRY2 DO
           <SOS>
         END ENTRY2;
         <SOS>

     --AS MANY 'OR' AND ACCEPT CLAUSES AS NEEDED

      END SELECT;
    END LOOP;
 END T1;
--WAIT FOR EITHER ENTRY1 OR ENTRY2
```

SELECTIVE WAIT WITH ELSE (DON'T WAIT AT ALL)

```
TASK T1 IS
    ENTRY ENTRY1;
END T1;
.
.
.
TASK BODY T1 IS
  BEGIN
   LOOP
    SELECT
      ACCEPT ENTRY1 DO
         <SOS>
      END ENTRY1;
      <SOS>
    ELSE
      <SOS>
    END SELECT;
   END LOOP;
  END T1;
```

IF THERE IS NOT A CALLER WAITING RIGHT NOW,
DO THE ELSE PART.

SELECTIVE WAIT WITH ELSE, MULTIPLE
ACCEPTS

```
TASK T1 IS
     ENTRY ENTRY1;
     ENTRY ENTRY2;
END T1;


TASK BODY T1 IS
  BEGIN
    LOOP
      SELECT
         ACCEPT ENTRY1 DO
           <SOS>
         END ENTRY1;
         <SOS>
      OR
         ACCEPT ENTRY2 DO
           ...
      -- AS MANY 'OR' AND 'ACCEPT' CLAUSES AS NEEDED

      ELSE
         <SOS>;
      END SELECT;
    END LOOP;
  END T1;
```

SELECT WITH DELAY ALTERNATIVE
(WAIT A FINITE TIME)

```
TASK BODY T1 IS
BEGIN
  LOOP
    SELECT
      ACCEPT ENTRY1 DO....
    [OR
      ACCEPT ENTRY2.....]
    OR
      DELAY 15.0; --SECONDS
      <SOS>;
    END SELECT;
  END LOOP;
END T1;
```

IF ENTRY1 CALLED WITHIN 15 SECONDS, THEN YOU ACCEPT THE CALL. OTHERWISE, AFTER 15 SECONDS YOU WILL DO SOMETHING.

'DELAY' RULES

YOU MAY HAVE SEVERAL ALTERNATIVES WITH A DELAY STATEMENT.

SINCE DELAYS CAN BE STATIC, THE SHORTEST DELAY ALTERNATIVE WILL BE SELECTED.

ZERO AND NEGATIVE DELAYS ARE LEGAL.

YOU MAY NOT HAVE AN ELSE PART WITH A DELAY, SINCE THE DELAY WOULD NEVER BE ACCEPTED.

SELECT WITH DELAY ALTERNATIVE
(WAIT A FINITE TIME)

```
TASK BODY T1 IS
BEGIN
  LOOP
    SELECT
      ACCEPT ENTRY1 DO....
    [OR
      ACCEPT ENTRY2.....]
    OR
      DELAY <EXPRESSION>;
      <SOS>;
    OR
      DELAY <EXPRESSION>;
      <SOS>;

    --SHORTEST DELAY WILL GET CHOSEN

    END SELECT;
  END LOOP;
END T1;
```

GUARDS CAN BE USED ON ANY ACCEPT STATEMENT

```
. . .
. . .
. . .
   WHEN SOME_CONDITION =>
      ACCEPT ENTRY1 ......
```

IF THERE IS NO GUARD, THE ACCEPT STATEMENT IS SAID TO BE OPEN.

IF THERE IS A GUARD, AND THE WHEN CONDITION IS TRUE, THE ACCEPT IS ALSO OPEN.

FALSE GUARD STATEMENTS ARE SAID TO BE CLOSED.

OPEN ALTERNATIVES ARE CONSIDERED. IF THERE IS MORE THAN ONE, THEN ONE IS SELECTED ARBITRARILY.

IF THERE ARE NO OPEN ALTERNATIVES (AND NO ELSE PART), THE EXCEPTION PROGRAM_ERROR IS RAISED.

TERMINATION

WHEN A TASK HAS COMPLETED ITS SEQUENCE OF STATEMENTS, ITS STATUS IS COMPLETED

ADDITIONALLY, THERE IS AN OPTION THAT ALLOWS A TASK TO TERMINATE.

```
SELECT
  ACCEPT ENTRY1 DO .....
[OR
  ACCEPT ENTRY2 DO.....]
 OR
  TERMINATE;
END SELECT;
```

THIS MAY NOT BE USED WITH EITHER THE THE DELAY OR AN ELSE CLAUSE.

SINCE THIS IS USED ONLY WITH A 'WAIT FOREVER' TASK, THIS OPTION ALLOWS A TASK THAT IS WAITING FOREVER TO TERMINATE IF ITS PARENT IS ALSO READY TO QUIT.

REMEMBER....

Tasks are Non-deterministic

```
select
    accept ENTRY1;
or
    accept ENTRY2;
```

Might always take ENTRY1!!!!

KILLING A TASK

OFTEN, A 'TERMINATE' ALTERNATIVE IS NOT SUFFICIENT.

A PARENT MAY KILL DEPENDENT TASKS (OR ITSELF) USING THE ABORT STATEMENT.

THIS SHOULD ONLY BE USED IN VERY RARE CIRCUMSTANCES.

A BETTER METHOD IS TO USE AN ENTRY TO 'ACCEPT' A SHUTDOWN CALL.

IF YOU HAVE ACCEPTED A 'SHUTDOWN' CALL, THEN IT IS OK TO ABORT YOURSELF.

```
TASK BODY T1 IS
BEGIN
 LOOP      -- THE ENDLESS LOOP OF THE
           --  TASK STARTS HERE
           -- EXIT LOOP TO TERMINATE
   SELECT
           -- THE REQUIRED ACCEPT
           --  STATEMENTS ARE CODED HERE
   OR
           ACCEPT SHUTDOWN;
           --SPECIAL FINAL ACTIONS HERE
           EXIT; -- EXITS LOOP, ENDS TASK
   OR
           TERMINATE; -- FOR CASES WHERE
                 -- SHUTDOWN NOT CALLED
   END SELECT;
 END LOOP;
END T1;
```

PROBLEMS WITH PARALLELISM

MULTIPLE 'THREADS OF CONTROL' CAN CAUSE PROBLEMS IF TWO PROCESSES ARE TRYING TO ACCESS AND UPDATE ONE PIECE OF INFORMATION AT THE SAME TIME.

PRAGMA SHARED

```
    MY-OBJECT : SOME-TYPE;
    PRAGMA SHARED (MY-OBJECT);
```

ENFORCES MUTUALLY EXCLUSIVE ACCESS

ONLY WORKS FOR SCALAR AND ACCESS TYPES

SEMAPHORES CAN ALSO BE USED TO CONTROL ACCESS TO AN OBJECT

-PROMOTES 'POLLING'

ENCAPSULATING A DATA ITEM WITHIN A TASK IS A BETTER METHOD

```
TASK SEMAPHORE IS
    ENTRY P; --GET RESOURCE
    ENTRY V; --RELEASE
END SEMAPHORE;

TASK BODY SEMAPHORE IS
     AVAILABLE : BOOLEAN := TRUE;
BEGIN
    LOOP
        SELECT
            WHEN AVAILABLE
            ACCEPT P DO
                AVAILABLE := FALSE;
            END P;
        OR
            WHEN NOT AVAILABLE
            ACCEPT V DO
                AVAILABLE := TRUE;
            END V;
        OR
            TERMINATE;
    END LOOP;
END SEMAPHORE;
```

```
TASK SPECIAL_OPS IS
   ENTRY ASSIGN ( OBJECT : IN SOME_TYPE );
   ENTRY RETRIEVE ( OBJECT : OUT SOME_TYPE);
END SPECIAL_OPS;

 TASK BODY SPECIAL_OPS IS
     THE_OBJECT : SOME_TYPE;
     BEGIN
        LOOP
          SELECT
           ACCEPT ASSIGN(OBJECT:IN SOME_TYPE)DO
              THE_OBJECT := OBJECT;
            END ASSIGN;
         OR
           ACCEPT RETRIEVE(OBJECT:OUT SOME_TYPE)DO
              OBJECT := THE_OBJECT;
           END RETRIEVE;
         OR
           TERMINATE;
          END SELECT;
        END LOOP;
END SPECIAL_OPS;
```

CALLING A TASK ENTRY

WHEN YOU CALL A TASK, YOU MUST KNOW THE TASK NAME.

THERE ARE THREE TYPES

- ENTRY CALLS (WAIT FOREVER)
- TIMED ENTRY CALLS (WAIT FOR SPECIFIED TIME)
- CONDITIONAL ENTRY CALLS (DON'T WAIT AT ALL)

CALL AND WAIT FOREVER

TO CALL AN ENTRY, SPECIFY THE TASK NAME AND THEN THE ENTRY NAME

```
BEGIN
 ...
  T1.ENTRY1(DATA);
```

TIMED ENTRY CALL
(WAIT FOR A FINITE TIME)

```
SELECT
   T1.ENTRY1(DATA);
      <SOS>
OR
   DELAY 60;
   <SOS>
END SELECT;
```

YOU CANNOT USE AN 'OR' TO CALL TWO (OR MORE) TASK ENTRIES!!!

THIS WOULD BE EQUIVALENT TO STANDING IN TWO DIFFERENT LINES AT ONCE.

CONDITIONAL ENTRY CALLS
(DON'T WAIT AT ALL)

```
SELECT
   T1.ENTRY1(DATA);
   <SOS>
ELSE
   <SOS>
END SELECT;
```

NOTICE THE 'ORTHOGONALITY' OR THE SELECT STATEMENT. IT IS USED IN EITHER A TASK ENTRY CALL OR AN ACCEPT STATEMENT.

ALSO NOTICE THAT INSTEAD OF
'ACCEPT...BEGIN...END ACCEPT;
IT IS
'ACCEPT...DO....END ENTRY_NAME;

WHY???

TASK ATTRIBUTES

T'CALLABLE - RETURNS BOOLEAN VALUE
TRUE -TASK CALLABLE,
FALSE -TASK COMPLETED,
ABNORMAL OR TERMINATED

T'TERMINATED - BOOLEAN VALUE
TRUE IF TERMINATED

E'COUNT - RETURNS AN UNIVERSAL
INTEGER INDICATING THE
NUMBER OF ENTRY CALLS
QUEUED FOR ENTRY E.

-AVAILABLE ONLY WITHIN
TASK T ENCLOSING E

TASK PRIORITIES

PRAGMA PRIORITY (STATIC_EXPRESSION);

USED TO REPRESENT DEGREE OF RELATIVE URGENCY.

IF TWO TASKS ARE READY, THEN THE TASK WITH THE HIGHER PRIORITY RUNS.

ALTHOUGH PRIORITIES ARE STATIC, TASK RENDEZVOUS ARE DYNAMIC. WHEN TASKS ARE IN RENDEZVOUS, THE PRIORITY IS THE HIGHER OF THE CALLER AND THE CALLEE.

SYNCHRONIZATION OF DATA

```
TASK SYNC IS
   ENTRY UPDATE ( DATA : IN DATA_TYPE);
   ENTRY READ   ( DATA :OUT DATA_TYPE);
END SYNC;

TASK BODY SYNC IS
  LOCAL : DATA_TYPE;
  BEGIN
    LOOP

      SELECT
         ACCEPT UPDATE(DATA : IN DATA_TYPE) DO
            LOCAL := DATA;
         END UPDATE;
      OR
         TERMINATE;
      END SELECT;

      SELECT
         ACCEPT READ (DATA : OUT DATA_TYPE) DO
            DATA := LOCAL;
         END READ;
      OR
         TERMINATE;
      END SELECT;

    END LOOP;
  END SYNC;
```

FAMILIES OF ENTR:ES

```
TYPE URGENCY IS (LOW, MEDIUM, HIGH);

TASK MESSAGE IS
     ENTRY RECEIVE(URGENCY) (DATA : DATA_TYPE);
END MESSAGE;

TASK BODY MESSAGE IS
  BEGIN
    LOOP
      SELECT
        ACCEPT RECEIVE(HIGH) (DATA:DATA_TYPE) DO
          ...
        END RECEIVE;
      OR
        WHEN RECEIVE(HIGH)'COUNT = 0 =>
        ACCEPT RECEIVE(MEDIUM) (DATA:DATA_TYPE) DO
          ...
        END RECEIVE;
      OR
        WHEN RECEIVE(HIGH)'COUNT+RECEIVE(MEDIUM)'COUNT=0 =>
        ACCEPT RECEIVE(LOW) (DATA:DATA_TYPE) DO
          ...
        END RECEIVE;
      OR
        DELAY 1.0; -- SHORT WAIT
   END MESSAGE;
```

SAME THING, WITH NO GUARDS

```
TYPE URGENCY IS (LOW, MEDIUM, HIGH);

TASK MESSAGE IS
    ENTRY RECEIVE(URGENCY) (DATA : DATA_TYPE);
END MESSAGE;

TASK BODY MESSAGE IS
  BEGIN
    LOOP
      SELECT
            ACCEPT RECEIVE(HIGH) (DATA:DATA_TYPE) DO
            ...
            END RECEIVE;
      ELSE
            SELECT
               ACCEPT RECEIVE(MEDIUM) (DATA:DATA_TYPE) DO
               ...
               END RECEIVE;
            ELSE
               SELECT
                  ACCEPT RECEIVE(LOW) (DATA:DATA_TYPE) DO
                  ...
                  END RECEIVE;
               OR
                  DELAY 1.0; -- SHORT WAIT
             END SELECT;
          END SELECT;
      END SELECT;
   END MESSAGE;
```

REPRESENTATION SPECIFICATIONS

LENGTH CLAUSE

T'STORAGE_SIZE

```
TASK TYPE T1 IS
    ENTRY ENTRY_1;
    FOR T1'STORAGE_SIZE USE
            2000*SYSTEM.STORAGE_UNIT);
END T1;
```

THE PREFIX T DENOTES A TASK TYPE.

THE SIMPLE EXPRESSION MAY BE STATIC, AND IS USED TO SPECIFY THE NUMBER OS STORAGE UNITS TO BE RESERVED OR FOR EACH ACTIVATION (NOT THE CODE) OF THE TASK.

ADDRESS CLAUSE

```
TASK TYPE T1 IS
     ENTRY ENTRY_1;
     FOR T1 USE AT 16#167A#;
END T1;
```

IN THIS CASE, THE ADDRESS SPECIFIES THE ACTUAL LOCATION IN MEMORY WHERE THE MACHINE CODE ASSOCIATED WITH T1 WILL BE PLACED.

```
TASK T1 IS
     ENTRY ENTRY_1;
     FOR ENTRY_1 USE AT 16#40#;
END T1;
```

IF THIS CASE, ENTRY_1 WILL BE MAPPED TO HARDWARE INTERRUPT 64.

ONLY IN PARAMETERS CAN BE ASSOCIATED WITH INTERRUPT ENTRIES.

AN INTERRUPT WILL ACT AS AN ENTRY CALL ISSUED BY THE HARDWARE, WITH A PRIORITY HIGHER THAN ANY USER-DEFINED TASK.

DEPENDING UPON THE IMPLEMENTATION, THERE CAN BE MANY RESTRICTIONS UPON THE TYPE OF CALL TO THE INTERRUPT, AND UPON THE TERMINATE ALTERNATIVES.

NOTE: YOU CAN DIRECTLY CALL AN INTERRUPT ENTRY.

TASKS AT DIFFERENT PRIORITIES

GIVEN 5 TASKS, 3 OF VARYING PRIORITY, 1 TO BE INTERRUPT DRIVEN, AND 1 THAT WILL BE TIED TO THE CLOCK.

```
PROCEDURE HEAVY_STUFF IS

   TASK HIGH_PRIORITY IS
        PRAGMA PRIORITY(50);  --OR AS HIGH AS SYSTEM ALLOWS
        ENTRY POINT;
   END HIGH_PRIORITY;

   TASK MEDIUM_PRIORITY IS
        PRAGMA PRIORITY(25);
        ENTRY POINT;
   END MEDIUM_PRIORITY;

   TASK LOW_PRIORITY IS
        PRAGMA PRIORITY(1);
        ENTRY POINT;
   END LOW_PRIORITY;

   TASK INTERRUPT_DRIVEN IS
        ENTRY POINT;
        FOR POINT USE AT 16#61#; --INTERRUPT 97
   END INTERRUPT_DRIVEN;

   TASK CLOCK_DRIVEN IS
        --THERE ARE TWO WAYS TO DO THIS

        --FIRST WAY IS TO HAVE ANOTHER TASK MONITOR
        -- THE CLOCK, AND CALL CLOCK_DRIVEN.CALL
        -- EVERY TIME UNIT.
        ENTRY CALL;

        --SECOND WAY IS TO ACTUALLY TIE CALL TO AN
        -- CLOCK INTERRUPT, AND LET CALL DETERMINE WHEN
        -- HE WISHES TO PERFORM AN ACTION
        FOR CALL USE AT 16#32#; --ASSUME INTERRUPT 50
                                -- IS A CLOCK INTERRUPT
     END CLOCK_DRIVEN;
END HEAVY_STUFF;
```

```
TASK QUEUE IS
    ENTRY INSERT(DATA : IN DATA_TYPE);
    ENTRY REMOVE(DATA :OUT DATA_TYPE);
END QUEUE;

TASK BODY QUEUE IS
   HEAD,TAIL : INTEGER := 0;
   Q : ARRAY (1..100) OF DATA_TYPE;
   BEGIN
     LOOP
       SELECT
         WHEN TAIL - HEAD + 1 /= 0 AND THEN
              TAIL - HEAD + 1 /= 100 =>
         ACCEPT INSERT(DATA : IN DATA_TYPE) DO
              IF HEAD = 0 THEN HEAD := 1; END IF;
              IF TAIL = 100 THEN TAIL := 0; END IF;
              TAIL := TAIL + 1;
              Q(TAIL) := DATA;
         END INSERT;
       OR
         WHEN HEAD /= 0 =>
         ACCEPT REMOVE(DATA :OUT DATA_TYPE) DO
              DATA := Q(HEAD);
              IF HEAD = TAIL THEN
                 HEAD := 0;
                 TAIL := 0;
              ELSE
                 HEAD := HEAD + 1;
                 IF HEAD > 100 THEN HEAD := 1; END IF;
              END IF;
          END REMOVE;
       OR
          TERMINATE;
       END SELECT;
     END LOOP;
   END QUEUE;
```

```
TASK TYPE QUEUE IS
    ENTRY INSERT(DATA : IN DATA_TYPE);
    ENTRY REMOVE(DATA :OUT DATA_TYPE);
END QUEUE;

TASK BODY QUEUE IS
   HEAD,TAIL : INTEGER := 0;
   Q : ARRAY (1..100) OF DATA_TYPE;
   BEGIN
     LOOP
       SELECT
         WHEN TAIL - HEAD + 1 /= 0 AND THEN
              TAIL - HEAD + 1 /= 100 =>
         ACCEPT INSERT(DATA : IN DATA_TYPE) DO
              IF HEAD = 0 THEN HEAD := 1; END IF;
              IF TAIL = 100 THEN TAIL := 0; END IF;
              TAIL := TAIL + 1;
              Q(TAIL) := DATA;
         END INSERT;
       OR
         WHEN HEAD /= 0 =>
         ACCEPT REMOVE(DATA :OUT DATA_TYPE) DO
              DATA := Q(HEAD);
              IF HEAD = TAIL THEN
                 HEAD := 0;
                 TAIL := 0;
              ELSE
                 HEAD := HEAD + 1;
                 IF HEAD > 100 THEN HEAD := 1; END IF;
              END IF;
          END REMOVE;
       OR
          TERMINATE;
       END SELECT;
     END LOOP;
   END QUEUE;

   MY_QUEUE, YOUR_QUEUE : QUEUE; -- TWO TASKS
```

```
GENERIC
DATA_TYPE : PRIVATE;
QUEUE_SIZE: POSITIVE := 100;

PACKAGE QUEUE_PACK IS

TASK QUEUE IS
    ENTRY INSERT(DATA : IN DATA_TYPE);
    ENTRY REMOVE(DATA :OUT DATA_TYPE);
END QUEUE;

PACKAGE BODY QUEUE_PACK IS
TASK BODY QUEUE IS
   HEAD,TAIL : INTEGER := 0;
   Q : ARRAY (1..QUEUE_SIZE) OF DATA_TYPE;
   BEGIN
     LOOP
       SELECT
         WHEN TAIL - HEAD + 1 /= 0 AND THEN
              TAIL - HEAD + 1 /= QUEUE_SIZE =>
         ACCEPT INSERT(DATA : IN DATA_TYPE) DO
              IF HEAD = 0 THEN HEAD := 1; END IF;
              IF TAIL = QUEUE_SIZE THEN TAIL := 0; END IF;
              TAIL := TAIL + 1;
              Q(TAIL) := DATA;
         END INSERT;
       OR
         WHEN HEAD /= 0 =>
         ACCEPT REMOVE(DATA :OUT DATA_TYPE) DO
              DATA := Q(HEAD);
              IF HEAD = TAIL THEN
                 HEAD := 0;
                 TAIL := 0;
              ELSE
                 HEAD := HEAD + 1;
                 IF HEAD> QUEUE_SIZE THEN HEAD := 1; END IF;
              END IF;
          END REMOVE;
       OR
          TERMINATE;
       END SELECT;
     END LOOP;
   END QUEUE;


PACKAGE NEW_QUEUE IS NEW QUEUE_PACK(MY_RECORD, 250);
PACKAGE OLD_QUEUE IS NEW QUEUE_PACK(INTEGER);
```

```
PROCEDURE INSERT_INTEGER (DATA : IN INTEGER ) RENAMES
    OLD_QUEUE.INSERT;

PROCEDURE REMOVE_INTEGER (DATA :OUT INTEGER ) RENAMES
    OLD_QUEUE.REMOVE;
```

```
PROCEDURE SPIN (R : RESOURCE) IS
BEGIN
  LOOP
    SELECT
       R.SEIZE;
       RETURN;
    ELSE
       NULL;  --BUSY WAITING
    END SELECT;
  END LOOP;
END;
```

--OR--

```
PROCEDURE SPIN (R : RESOURCE) IS
BEGIN
   R.SEIZE;
   RETURN;
END;
```

ADA TASKING

SCENARIO I

"THE GOLDEN ARCHES"

MCD TASKS :

- SERVICE PROVIDED : FOOD
- SERVICE REQUESTED : NONE

GONZO TASKS :

- SERVICE PROVIDED : NONE
- SERVICE REQUESTED : FOOD

```
Task McD is
    entry SERVE(TRAY_OF : out FOOD_TYPE);
end McD;

Task GONZO;

Task Body McD is
    NEW_TRAY : FOOD_TYPE;
    function COOK return FOOD_TYPE is ......
    begin
        loop
            accept SERVE(TRAY_OF : out FOOD_TYPE) do
                TRAY_OF := COOK;
            end;
        end loop;
    end McD;
```

```
Task Body GONZO is
    MY_TRAY : FOOD_TYPE;

    procedure CONSUME(MY_TRAY:in FOOD_TYPE) is ...

    begin
        loop
            McD.SERVE ( MY_TRAY);
            CONSUME(MY_TRAY);
        end loop;
    end GONZO;
```

```
Task Body McD is
    NEW_TRAY : FOOD_TYPE;

    function COOK return FOOD_TYPE is
        ...
    end COOK;

    begin
        loop
            NEW_TRAY := COOK;
            accept SERVE(TRAY_OF:out FOOD_TYPE) do
                TRAY_OF := NEW_TRAY;
            end SERVE;
        end loop;
    end GONZO;
```

```
loop
   NEW_TRAY := COOK;
   select
      accept SERVE(TRAY_OF : out FOOD_TYPE) do
         TRAY_OF := NEW_TRAY;
      end SERVE;
   else
      null;
   end select;
end loop;
```

```
loop
    NEW_TRAY := COOK;
    select
        accept SERVE(TRAY_OF : out FOOD_TYPE) do
            TRAY_OF := NEW_TRAY;
        end SERVE;
    else or
        terminate;
    end select;
end loop;
```

```
loop
    NEW_TRAY := COOK;
    select
        accept SERVE(TRAY_OF : out FOOD_TYPE) do
            TRAY_OF := NEW_TRAY;
        end SERVE;
    or
        delay 15.0 * MINUTES;
    end select;
end loop;
```

```
loop

    select
        McD.SERVE(MY_ORDER);  CONSUME(MY_ORDER);
    else
        select
            BK.SERVE(MY_ORDER); CONSUME (MY_ORDER);
        else
            exit;
        end select;
    end select;

end loop;
```

```
loop

    select
        McD.SERVE(MY_ORDER);  CONSUME(MY_ORDER);
    or
        delay 5.0 * MINUTES;
        select
            BK.SERVE(MY_ORDER); CONSUME (MY_ORDER);
        or
            delay 5.0 * MINUTES;
            exit;
        end select;
    end select;

end loop;
```

```
loop

    select
        McD.SERVE (MY_ORDER);
    or                                              --**
        BK.SERVE(MY_ORDER);
    end select;

    CONSUME(MY_ORDER);

end loop;
```

```
loop

    select
        McD.SERVE (MY_ORDER);                    --**
    or
        BK.SERVE(MY_ORDER);
    else
        delay 10.0 * MINUTES;
        exit;
    end select;

    CONSUME(MY_ORDER);

end loop;
```

ADA TASKING

SCENARIO II

"NO FREE LUNCH"

MCD TASK
- SERVICE PROVIDED : FOOD
- SERVICE REQUESTED: MONEY

GONZO TASK
- SERVICE PROVIDED : MONEY
- SERVICE REQUESTED: FOOD

```
Task McD is
    entry SERVE(ORDER: out FOOD_TYPE;
                        COST: in MONEY_TYPE);
end McD;

Task GONZO;



--OR

Task McD is
    entry SERVE(ORDER: out FOOD_TYPE);
end McD;

Task GONZO is
    entry PAY (COST : in MONEY_TYPE;
               PAYMENT : out MONEY_TYPE);
end GONZO;
```

+

```
Task Body McD is
   CASH_DRAWER, AMOUNT_PAID: MONEY_TYPE;
   NEW_ORDER  : FOOD_TYPE;
   function COOK ..........
   function CALC_COST(ORDER: in FOOD_TYPE)
      return MONEY_TYPE ..........

   begin
      loop
         NEW_ORDER := COOK;
         select
            accept SERVE(ORDER:out FOOD_TYPE) do
               ORDER := NEW_ORDER;
               COST := CALC_COST(NEW_ORDER);
               GONZO.PAY(COST, AMOUNT_PAID); --**
               CASH_DRAWER :=
                  CASH_DRAWER + AMOUNT_PAID;
            end SERVE;
         or
            delay 15.0 * MINUTES;
         end select;
      end loop;
   end McD;
```

```
Task Body GONZO IS
    ACCOUNT_BALANCE : MONEY_TYPE;
    MY_ORDER : FOOD_TYPE;
    function GO_TO_WORK return MONEY_TYPE ......

    begin
        ACCOUNT_BALANCE :=
            ACCOUNT_BALANCE + GO_TO_WORK;
        loop
            McD.SERVE(MY_ORDER);
            accept PAY (COST : in MONEY_TYPE;
                        PAYMENT:out MONEY_TYPE) do
                ACCOUNT_BALANCE :=
                    ACCOUNT_BALANCE - COST;
                PAYMENT := COST;
            end PAY;
        end loop;
    end GONZO;
```

+

Scenario II a

"No Wait for the Waiters"

McD Task

Service Provided : Food
Service Requested: Money

Gonzo Task

Service Provided : Money
Service Requested: Food

Manager Task

Service Provided : Make new Waiter
Service Requested: None

```
Task type McD is
    entry SERVE.....
end McD;

Task GONZO is
    entry PAY.....
end GONZO;

Task MANAGER;

Type CASHIER_POINTER is access McD;

Type REGISTER_TYPE is array (1..NO_REGISTERS)
                of CASHIER_POINTER;

THE_REGISTERS := REGISTER_TYPE
                    := (others => new McD);
```

```
Task Body McD is
    ...
    ...
    ...
    begin
        loop
            NEW_ORDER := COOK;
            select
                accept SERVE.....
                ...
                end SERVE;
            or
                delay 2,0 * MINUTES;
                exit;
            end select;
        end loop;
```

```
Task Body GONZO is
    ...
    ...
    begin
        ...
        ...
        --- Now, GONZO has to search for the open
        ---  registers, and select the one with
        ---  the shortest line
        ...
        ...
        THE_REGISTERS(MY_REGISTER).SERVE...
        ...
    end GONZO;
```

```
Task Body MANAGER is
    ...
    ...
    begin
        loop
        --The Manager will look at the queue lengths of
        --  the open registers, and, when necessary,
        --  will open registers that are currently
        --  closed
        ...
        ...
            if ...........then
                THE_REGISTERS(CLOSED_REGISTER):=
                                                new McD;
            end if;
        end loop;
    end MANAGER;
```

```
Task BR is
    entry SERVE(ICE_CREAM: out DESSERT_TYPE;
end BR;

Task SERVOMATIC is
    entry TAKE(A_NUMBER: out SERVOMATIC_NUMBERS);
end SERVOMATIC;

Task type CUSTOMER_TASK is
    entry REQUEST(ORDER: out ORDER_TYPE);
enter CUSTOMER_TASK;

Type CUSTOMER is access CUSTOMER_TASK;

CUSTOMERS : array (SERVOMATIC_NUMBERS) of CUSTOMER;
```

ADA TAS' ING

SCENARIO III

"A SUGAR CONE, PLEASE:

```
Task Body BR is
   NEXT_CUSTOMER : SERVOMATIC_NUMBERS :=
                          SERVOMATIC_NUMBERS'last;
   CUREENT_ORDER : ORDER_TYPE;
   ICE_CREAM : DESSERT_TYPE;
   function MAKE(ORDER : in ORDER_TYPE) return
                      DESSERT_TYPE is ......
   begin
     loop
        begin
          NEXT_CUSTOMER:=(NEXT_CUSTOMER+1)
                      mod SERVOMATIC_NUMBERS'last;
          CUSTOMERS(NEXT_CUSTOMER).REQUEST
                                  (CURRENT_ORDER);
          ICE_CREAM := MAKE (CURRENT_ORDER);
          accept SERVE(ICE_CREAM:out DESSERT_TYPE) do
             ICE_CREAM := BR.ICE_CREAM;
          end SERVE;
        exception
          when TASKING_ERROR=>null;--customer not here
        end;
     end loop
   end;
```

```
Task Body SERVOMATIC is
    NEXT_NUMBER : SERVOMATIC_NUMBERS :=
                        SERVOMATIC_NUMBERS'first;

    begin
      loop
        accept TAKE(A_NUMBER:out SERVOMATIC_NUMBERS)d
            A_NUMBER := NEXT_NUMBER;
        end TAKE;
        NEXT_NUMBER:=(NEXT_NUMBER + 1) mod
                              SERVOMATIC_NUMBERS'last;
      end loop;

    end SERVOMATIC;
```

```
Task Body CUSTOMER_TASK is
    MY_ORDER : ORDER_TYPE := ...  -- some value
    MY_DESSERT : DESSERT_TYPE;

    begin
        accept REQUEST(ORDER:out ORDER_TYPE) do
            ORDER := MY_ORDER;
        end REQUEST;
        BR.SERVE(MY_DESSERT);
        --eat the dessert, or do whatever
    end;
```

Ada Tasking

Scenario IV

"Lets Hide the Spooler Task"

PRINTER_PACKAGE
Action-"Hides" the print spooler
by renaming task entry

SPOOLER Task
Service Provided : Virtual Print
Service Requested: Physical Print

PRINTER Task
Service Provided : Physical Print
Service Requested: File Name

```
Package PRINTER_PACKAGE is
    ...
    ...
    task SPOOLER is
        entry PRINT_FILE(NAME : in STRING;
                         PRIORITY : in NATURAL);
        entry PRINTER_READY;
    end SPOOLER;
    ...
    ...
    procedure PRINT (NAME : in STRING;
                     PRIORITY : in NATURAL := 10)
          renames SPOOLER.PRINT_FILE;
    end PRINTER_PACKAGE;


    Package Body PRINTER_PACKAGE is
        ...
        task PRINTER is
            entry PRINT_FILE(NAME : in STRING);
        end PRINTER;
        ...
    end PRINTER_PACKAGE;
```

```
Task Body SPOOLER is
    begin
        loop
            select
                accept PRINTER_READY do
                    PRINTER.PRINT_FILE(REMOVE(QUEUE));
                    --Remove would determine the next job
                    -- and send it to the actual printer
                end PRINTER_READY;
            else
                null;
            end select;

            select
                accept PRINT_FILE(NAME : in STRING;
                                  PRIORITY : NATURAL ) do
                    INSERT (NAME, PRIORITY);
                    --put name on queue or queues
                    --  according to priority
                end PRINT_FILE;
            else
                null;
            end select;
        end loop;
    end SPOOLER:
```

```
Task Body PRINTER is
    begin
        loop
            SPOOLER.PRINTER_READY;
            accept PRINT_FILE (NAME : in STRING) do

                if NAME'length /= 0 then .....

                    -- print the file



                else
                    delay 10.0 * SECONDS;
                 end if;

            end PRINT_FILE;
        end loop;

    end PRINTER;
```

```
with PRINTER_PACKAGE;

procedure MAIN is
    ...
    ...
    ...
    loop
        -- process several files

        PRINTER_PACKAGE.PRINT (A_FILE, A_PRIORITY);
        ...
        ...
    end loop;
end MAIN;
```

+

TASKING MINDSET

SIMPLE PROBLEM - WRITE A TASK SPEC TO LET TASK A SEND AN INTEGER TO TASK B.

SOLUTION 1 - A CALLS AN ENTRY IN B

SOLUTION 2 - B CALLS FOR AN ENTRY IN A

SOLUTION 3 - WRITE A 'BUFFER' TASK
TO CALL ENTRY IN A, GET INTEGER, AND
THEN CALL ENTRY IN B TO SEND INTEGER

SOLUTION 4 - WRITE BUFFER TASK C TO ACCEPT INTEGERS FROM A, AND ALSO ACCEPT REQUESTS FROM B

IN-CLASS EXERCISE

LET US DESIGN THE TASK SPECIFICATIONS FOR THE FOLLOWING SENARIO.

THREE TASKS HAVE ACCES TO A TYPE KNOWN AS MESSAGE_TYPE.

TASK_1 PRODUCES MESSAGES. TASK_2 CAN RECEIVE MESSAGES, HOLD THEM IN A BUFFER (IF NECESSARY), AND SENDS THEM TO TASK_3 WHEN THE DATE/TIME FIELD (PART OF MESSAGE_TYPE) SAYS TO.

```
TASK TASK_1 IS



END TASK_1;



TASK TASK_2 IS



END TASK_2;



TASK TASK_3 IS



END TASK_3;
```

TASKING EXERCISE

WRITE A MAIN PROGRAM AND TWO TASKS TO SIMULATE A HOUSE ALARM SYSTEM. THE MAIN PROGRAM IS AN INPUT SIMULATOR TO THE TASKS. ONE TASK KEEPS TRACK OF THE STATUS OF THE HOUSE. ANOTHER IS THE ACTUAL ALARM SYSTEM.

TASK 1: THE HOUSE STATUS (TASK NAME :HOUSE)
THREE ENTRIES => OK, NOT_OK, WRITE

THE ENTRIES OK AND NOT_OK SET OR RESET A FLAG THAT DETERMINES THE STATUS OF THE HOUSE. NOT_OK WILL ALSO SET A VARIABLE TO TELL YOU WHICH ALARM IS CURRENTLY GOING OFF. BOTH OK AND NOT_OK SHOULD PRINT OUT A MESSAGE VERIFYING THAT THEY WERE CALLED. THE WRITE ENTRY WILL PRINT THE STATUS OF THE HOUSE. IF THERE IS AN ALARM CURRENTLY GOING OFF, WRITE WILL TELL YOU THE ALARM NUMBER.

TASK 2: THE ALARM SYSTEM (TASK NAME: ALARM)
THREE ENTRIES => FIRE, INTRUDER, SHUTOFF

THE ALARM SYSTEM WILL ACCEPT ANY OF THE THREE ENTRY CALLS FROM THE INPUT SIMULATOR. IF THERE ARE NO ENTRY CALLS WITHIN 5 SECONDS, IT WILL CALL HOUSE.WRITE TO DISPLAY THE STATUS. FIRE AND INTRUDER EACH HAVE A PARAMETER INDICATION THE ALARM LOCATION. FIRE LOCATIONS ARE '1' THRU '9'. INTRUDER LOCATIONS ARE 'A' THRU 'Z'. FIRE AND INTRUDER SHOULD CALL HOUSE.NOT_OK (AND TELL THE HOUSE WHERE THE ALARM IS SOUNDING), AND THEN PRINT OUT A MESSAGE

MAIN PROGRAM

THE MAIN PROGRAM WILL READ IN CHARACTERS FROM THE KEYBOARD. IF THE CHARACTER IS A '1' THRU '9', CALL THE FIRE ALARM. IF THE CHARACTER IS A 'A' THRU 'Z' THEN IT CALLS THE INTRUDER ALARM. IF THE CHARACTER IS A '0'(ZERO), THE HOUSE IS RESET TO OK. IF THE CHARACTER IS A '!', THEN THE ALARM IS SHUTDOWN, AND THE PROGRAM ENDS. ALL OTHER CHARACTERS DO NOTHING.

THE HOUSE STATUS SHOULD BE OK TO START.

```
run cookie


The house is ok


The house is ok

&
Invalid character.  Try again


The house is ok

G
House alarm set to not OK at location G
Intruder in room G

The house is not ok ..alarm is off at location G


The house is not ok ..alarm is off at location G

4
House alarm set to not OK at location 4
Fire Alarm # 4 has been set off.

The house is not ok ..alarm is off at location 4

Ø
House alarm reset to OK.

The house is ok


The house is ok

!
The alarm has been turned off

*)
```

```
WI.H TEXT_IO;
USE TEXT_TO;

PROCEDURE COOKIE  IS

CHAR : CHARACTER;

    TASK HOUSE  IS
        ENTRY OK;
        ENTRY NOT_OK (WHERE:CHARACTER);
        ENTRY WRITE;
    END HOUSE ;

    TASK ALARM  IS
        ENTRY FIRE     (LOCATION:CHARACTER);
        ENTRY INTRUDER (LOCATION:CHARACTER);
        ENTRY SHUTOFF;
    END ALARM ;
```

```
TASK BODY HOUSE  IS
    TYPE CONDITION IS (OK, NOT_OK);
    ALARM_STATUS : CONDITION := OK;
    ALARM_LOCATION : CHARACTER;

BEGIN
    LOOP
        SELECT
            ACCEPT OK DO
                ALARM_STATUS := OK;
                PUT_LINE("House ALARM RESET TO OK.");
            END OK;
        OR
            ACCEPT NOT_OK (WHERE:CHARACTER) DO
                ALARM_STATUS := NOT_OK;
                ALARM_LOCATION := WHERE;
                PUT_LINE("House ALARM SET TO NOT OK AT"&
                    "LOCATION " & ALARM_LOCATION);
            END NOT_OK;
        OR
            ACCEPT WRITE DO
                NEW_LINE;
                CASE ALARM_STATUS IS
                  WHEN OK =>PUT_LINE("THE HOUSE IS OK");
                  WHEN NOT_OK =>  PUT_LINE
                        ("THE HOUSE IS NOT OK"&
                         " ..ALARM IS OFF AT LOCATION " &
                          ALARM_LOCATION);
                END CASE;
                NEW_LINE;
            END WRITE;
        OR
            TERMINATE;
        END SELECT;
    END LOOP;
END HOUSE ;
```

```
TASK BODY ALARM  IS
BEGIN
    LOOP
        SELECT
            ACCEPT FIRE (LOCATION:CHARACTER) DO
                HOUSE.NOT_OK(LOCATION);
                PUT ("FIRE ALARM # ");
                PUT (LOCATION);
                PUT_LINE (" HAS BEEN SET OFF.");
            END FIRE;
        OR
            ACCEPT INTRUDER (LOCATION:CHARACTER) DO
                HOUSE.NOT_OK(LOCATION);
                PUT ("INTRUDER IN ROOM ");
                PUT (LOCATION);
                NEW_LINE;
            END INTRUDER;
        OR
            ACCEPT SHUTOFF;
            PUT_LINE ("THE ALARM HAS BEEN TURNED OFF");
            EXIT;
        OR
            DELAY 5.0;
            HOUSE.WRITE;
        END SELECT;
    END LOOP;
END ALARM;
```

```
BEGIN          --MAIN
    LOOP
        GET (CHAR);
        SKIP_LINE;
        CASE CHAR IS
            WHEN '1' .. '9' =>  ALARM.FIRE (CHAR);
            WHEN 'A' .. 'Z' =>  ALARM.INTRUDER (CHAR);
            WHEN 'A' .. 'Z' =>  ALARM.INTRUDER (CHAR);
            WHEN '0'        =>  HOUSE.OK;
            WHEN '!'        =>  ALARM.SHUTOFF;
            WHEN OTHERS     =>  PUT_LINE
                     ("INVALID CHARACTER.  TRY AGAIN");
        END CASE;
        EXIT WHEN CHAR = '!';
    END LOOP;

END COOKIE;
```

Tutorial on Ada Exceptions

by
Major Patricia K. Lawlis
lawlis%asu@csnet-relay

Air Force Institute of Technology (AFIT)
and
Arizona State University (ASU)

27 January 1989

References

- Student Handout, "Ada Applications Programmer - Advanced Ada Software Engineering", USAF Technical Training School, Keesler Air Force Base, July 1986.

- J. G. P. Barnes, Programming in Ada, Second edition, Addison-Wesley, 1984.

- Grady Booch, Software Engineering with Ada, Second Edition, Benjamin/Cummings, 1987.

- Putnam P. Texel, Introductory Ada: Packages for Programming, Wadsworth, 1986.

- Eugen N. Vasilescu, Ada Programming with Applications, Allyn and Bacon, 1986.

- ANSI/MIL-STD-1815A, "Military Standard - Ada Programming Language" (**LRM**), U. S. Department of Defense, 22 January 1983.

Outline

=> Overview

- Naming an exception
- Creating an exception handler
- Raising an exception
- Handling exceptions
- Turning off exception checking
- Tasking exceptions
- More examples
- Summary

Overview

- What is an exception
- Ada exceptions
- Comparison
  - the American way
  - using exceptions

What Is an Exception

- A run time error
- An unusual or unexpected condition
- A condition requiring special attention
- Other than normal processing
- An important feature for debugging
- A critical feature for operational software

Ada Exceptions

- An exception has a name
  - may be predefined
  - may be declared
- The exception is raised
  - may be raised implicitly by run time system
  - may be raised explicitly by **raise** statement
- The exception is handled
  - exception handler may be placed in any **frame***
  - exception propagates until handler is found
  - if no handler anywhere, process aborts

* executable part surrounded by begin - end

The American Way

```
package Stack_Package is

    type Stack_Type is limited private;

    procedure Push (Stack            : in out Stack_Type;
                    Element          : in     Element_Type;
                    Overflow_Flag    : out    BOOLEAN);

    ...

end Stack_Package;


with TEXT_IO;
with Stack_Package; use Stack_Package;
procedure Flag_Waving is
    ...
    Stack   : Stack_Type;
    Element : Element_Type;
    Flag    : BOOLEAN;
begin
    ...
    Push (Stack, Element, Flag);
    if Flag then
        TEXT_IO.PUT ("Stack overflow");
        ...
    end if;
    ...
end Flag_Waving;
```

Using Exceptions

```
package Stack_Package is

        type Stack_Type is limited private;
        Stack_Overflow,
        Stack_Underflow : exception;

        procedure Push (Stack      : in out   Stack_Type;
                        Element    : in       Element_Type);
                -- may raise Stack_Overflow
        ...
end Stack_Package;


with TEXT_IO;
with Stack_Package; use Stack_Package;
procedure More_Natural is
        ...
        Stack    : Stack_Type;
        Element : Element_Type;
begin
        ...
        Push (Stack, Element);
        ...
exception
        when Stack_Overflow =>
                TEXT_IO.PUT ("Stack overflow");
        ...
end More_Natural;
```

Outline

- Overview

=> **Naming an exception**

- Creating an exception handler
- Raising an exception
- Handling exceptions
- Turning off exception checking
- Tasking exceptions
- More examples
- Summary

Naming an Exception

- Predefined exceptions
- Declaring exceptions
- I/O exceptions

Predefined Exceptions

- In package STANDARD (also see chap 11 of LRM)

- CONSTRAINT_ERROR

  violation of range, index, or discriminant constraint...

- NUMERIC_ERROR

  execution of a predefined numeric operation cannot deliver a correct result

- PROGRAM_ERROR

  attempt to access a program unit which has not yet been elaborated...

- STORAGE_ERROR

  storage allocation is exceeded...

- TASKING_ERROR

  exception arising during intertask communication

Declaring Exceptions

exception_declaration ::= identifier_list : **exception**;

- Exception may be declared anywhere an object declaration is appropriate

- However, exception is not an object

  - may not be used as subprogram parameter, record or array component

  - has same scope as an object, but its effect may extend beyond its scope

Example:

```
procedure  Calculation  is

      Singular                    : exception;
      Overflow, Underflow         : exception;

begin
   --
end  Calculation;
```

I/O Exceptions

- Exceptions relating to file processing

- In predefined library unit IO_EXCEPTIONS
  (also see chap 14 of LRM)

- TEXT_IO, DIRECT_IO, and SEQUENTIAL_IO **with** it

```
package IO_EXCEPTIONS is

        NAME_ERROR    : exception;
        USE_ERROR     : exception;     --attempt to use
                                       --invalid operation
        STATUS_ERROR  : exception;
        MODE_ERROR    : exception;
        DEVICE_ERROR  : exception;
        END_ERROR     : exception;     --attempt to read
                                       --beyond end of file
        DATA_ERROR    : exception;     --attempt to input
                                       --wrong type
        LAYOUT_ERROR  : exception;     --for text processing

end IO_EXCEPTIONS;
```

Outline

- Overview
- Naming an exception

=> Creating an exception handler

- Raising an exception
- Handling exceptions
- Turning off exception checking
- Tasking exceptions
- More examples
- Summary

Creating an Exception Handler

- Defining an exception handler
- Restrictions
- Handler example

Defining an Exception Handler

- Exception condition is "caught" and "handled" by an exception handler

- Exception handler may appear at the end of any frame (block, subprogram, package or task body)

```
begin
    ...
exception
    -- exception handler(s)
end;
```

- Form similar to case statement

exception_handler ::=
    when exception_choice {| exception_choice} **=>**
        sequence_of_statements

exception_choice ::= exception_name | **others**

Restrictions

- Exception handlers must be at the end of a frame
- Nothing but exception handlers may lie between **exception** and **end** of frame
- A handler may name any visible exception declared or predefined
- A handler includes a sequence of statements
  - response to exception condition
- A handler for **others** may be used
  - must be the last handler in the frame
  - handles all exceptions not listed in previous handlers of the frame (including those not in scope of visibility)
  - can be the only handler in the frame

Handler Example

```
procedure Whatever is

        Problem_Condition : exception;

begin
        ...

exception

        when Problem_Condition =>
                Fix_It;

        when CONSTRAINT_ERROR =>
                Report_It;

        when others =>
                Punt;

end Whatever;
```

Outline

- Overview
- Naming an exception
- Creating an exception handler

=> **Raising an exception**

- Handling exceptions
- Turning off exception checking
- Tasking exceptions
- More examples
- Summary

Raising an Exception

- Elaboration and execution exceptions
- How exceptions are raised
- Effects of raising an exception
- Raising example

Elaboration and Execution Exceptions

- Elaboration exceptions occur when declarations are being elaborated
  - after a unit is "called"
  - before execution of the unit begins
  - can only be predefined exceptions
- Execution exceptions occur during execution of a frame
- Elaboration exceptions can also be considered as execution exceptions
  - depending on viewpoint
  - can consider as part of the execution of the last executable statement making the call to the unit being elaborated
  - this helps with understanding the consistency of the rules for exception handling

How Exceptions are Raised

- Implicitly by run time system
  - predefined exceptions

- Explicitly by **raise** statement

raise_statement ::= **raise** [exception_name];

  - the name of the exception must be visible at the point of the raise statement
  - a raise statement without an exception name is allowed only within an exception handler

Effects of Raising an Exception

(1) Control transfers to exception handler at end of frame being **executed** (if handler exists)

(2) Exception is lowered

(3) Sequence of statements in exception hander is executed

(4) Control passes to end of frame

- If frame does not contain an appropriate exception handler, the exception is propagated - effectively skipping steps 1 thru 3 and going straight to step 4

Raising Example

```
procedure Whatever is

        Problem_Condition        : exception;
        Real_Bad_Condition       : exception;

begin
        ...
        if Problem_Arises then
                raise Problem_Condition;                --  1
        end if;
        ...
        if Serious_Problem then
                raise Real_Bad_Condition;               --  1
        end if;
        ...
exception

        when Problem_Condition =>                       --  2
                Fix_It;                                 --  3

        when CONSTRAINT_ERROR =>                        --  2
                Report_It;                              --  3

        when others =>                                  --  2
                Punt;                                   --  3

end Whatever;                                           --  4
```

Outline

- Overview
- Naming an exception
- Creating an exception handler
- Raising an exception

=> **Handling exceptions**

- Turning off exception checking
- Tasking exceptions
- More examples
- Summary

Handling Exceptions

- How exception handling can be useful
- Which exception handler is used
- Sequence of statements in exception handler
- Propagation
- Propagation example

How Exception Handling Can Be Useful

- Normal processing could continue if
  - cause of exception condition can be "repaired"
  - alternative approach can be used
  - operation can be retried
- Degraded processing could be better than termination
  - for example, safety-critical systems
- If termination is necessary, "clean-up" can be done first

Which Exception Handler Is Used

- When exception is raised, system looks for an exception handler at the end of the frame being executed

- If exception is raised during elaboration of the declarative part of a unit (unit is not yet ready to execute)

  - elaboration is abandoned and control goes to the end of the unit with the exception still raised

  - exception part of the unit is not searched for an appropriate handler

  - effectively, the calling unit will be searched for an appropriate handler
    -- consistent with execution viewpoint

  - if elaboration of library unit, program execution is abandoned
    -- all library units are elaborated with the main program

- If exception is raised in exception handler

  - handler may contain block(s) with handler(s)

  - if not handled locally within handler, control goes to end of frame with exception raised

Sequence of Statements in Exception Handler

- Handler completes the execution of the frame
  - handler for a **function** should usually contain a **return** statement
- Statements can be of arbitrary complexity
  - can use most any language construct that makes sense in that context
  - cannot use **goto** statement to transfer into a handler
  - if handler is in a block inside a loop, could use **exit** statement
- Handler at end of package body applies only to package initialization

Propagation

- Occurs if no handler exists in frame where execution exception is raised
- Always occurs if elaboration exception is raised
- Also occurs if **raise** statement is used in handler
- Exception is propagated dynamically
  - propagates from subprogram to unit calling it (not necessarily unit containing its declaration)
  - this can result in propagation outside its scope
  - task propagation follows same principle, but a little more complicated
- Propagation continues until
  - an appropriate handler is found
  - exception propagates to main program (still with no handler) and program execution is abandoned

Propagation Example

```
procedure Do_Nothing is
    --------------
    procedure Has_It is
        Some_Problem : exception;
    begin
        ...
        raise Some_Problem;
        ...
    exception
        when Some_Problem =>
            Clean_Up;
            raise;
    end Has_It;
    --------------
    procedure Calls_It is
    begin
        ...
        Has_It;
        ...
    end Calls_It;
    --------------
begin  -- Do_Nothing
    ...
    Calls_It;
    ...
exception
    when others => Fix_Everything;
end Do_Nothing;
```

Outline

- Overview
- Naming an exception
- Creating an exception handler
- Raising an exception
- Handling exceptions

=> **Turning off exception checking**

- Tasking exceptions
- More examples
- Summary

Turning Off Exception Checking

- Overhead vs efficiency
- Pragma SUPPRESS
- Check identifiers

Overhead vs Efficiency

- Exception checking imposes run time overhead

  - interactive applications will never notice
  - real-time applications have legitimate concerns
    but must not sacrifice system safety

- When efficiency counts

  - first, make program work (using good design)
  - be sure possible problems are covered by exception handlers
  - check if efficient enough - stop if it is

  - if not, study execution profile
    - -- eliminate bottlenecks
    - -- improve algorithm
    - -- avoid "cute" tricks

  - check if efficient enough - stop if it is
  - if not, trade-offs may be necessary
  - some exception checks may be expendable since debugging is done

  - however, every suppressed check poses new possibilities for problems
    - -- must re-examine possible problems
    - -- must re-examine exception handlers

  - always keep in mind
    - -- problems <u>will</u> happen
    - -- critical applications must be able to deal with these problems

Moral

Improving the design is far better - and easier in the long run - than suppressing checks

Pragma SUPPRESS

- Only allowed immediately within a declarative part or immediately within a package specification

pragma SUPPRESS (identifier [,[**ON** =>] name]);

- identifier is that of the check to be omitted (next slide lists identifiers)
- name is that of an object, type, or unit for which the check is to be suppressed
  - -- if no name is given, it applies to the remaining declarative region

- An implementation is free to ignore the suppress directive for any check which may be impossible or too costly to suppress

Example:

pragma SUPPRESS (INDEX_CHECK, ON => Index);

Check Identifiers

- These identifiers are explained in more detail in chap 11 of the LRM

- Check identifiers for suppression of CONSTRAINT_ERROR checks

    ACCESS_CHECK
    DISCRIMINANT_CHECK
    INDEX_CHECK
    LENGTH_CHECK
    RANGE_CHECK

- Check identifiers for suppression of NUMERIC_ERROR checks

    DIVISION_CHECK
    OVERFLOW_CHECK

- Check identifier for suppression of PROGRAM_ERROR checks

    ELABORATION_CHECK

- Check identifier for suppression of STORAGE_ERROR check

    STORAGE_CHECK

Outline

- Overview
- Naming an exception
- Creating an exception handler
- Raising an exception
- Handling exceptions
- Turning off exception checking

=> Tasking exceptions

- More examples
- Summary

Tasking Exceptions

- Exception handling is trickier for tasks
- Exceptions during task communication
- Tasking example

Exception Handling Is Trickier for Tasks

- Rules are not really different, just more involved
  - local exceptions handled the same within frames

If exception is raised

- during elaboration of task declarations
  - the exception TASKING_ERROR will be raised at the point of task activation (becomes execution exception in enclosing subprogram)
  - the task will be marked completed
- during execution of task body (and not resolved there)
  - task is completed
  - exception is <u>not</u> propagated
- during task rendezvous
  - this is the really tricky part

Exceptions During Task Communication

- If the **called** task terminates abnormally

  exception TASKING_ERROR is raised in **calling** task at the point of the entry call

- If an entry call is made for entry of a task that becomes completed before accepting the entry

  exception TASKING_ERROR is raised in **calling** task at the point of the entry call

- If the **calling** task terminates abnormally

  no exception propagates to the **called** task

- If an exception is raised in **called** task within an **accept** (and not handled there locally)

  the same exception is raised in the **calling** task at the point of the entry call
  (even if exception is later handled outside of the accept in the called task)

Tasking Example

```
procedure Critical_Code is

        Failure : exception;
        -------------
        task Monitor is
                entry Do_Something;
        end Monitor;
        task body Monitor is
        ...
        begin
                accept Do_Something do
                        ...
                        raise Failure;
                        ...
                end Do_Something;
                ...
        exception  -- exception handled here
                when Failure =>
                        Termination_Message;
        end Monitor;
        -------------
begin -- Critical_Code
        ...
        Monitor.Do_Something;
        ...
exception  -- same exception will be handled here
        when Failure =>
                Critical_Problem_Message;

end Critical_Code;
```

Outline

- Overview
- Naming an exception
- Creating an exception handler
- Raising an exception
- Handling exceptions
- Turning off exception checking
- Tasking exceptions

=> **More examples**

- Summary

More Examples

- Interactive data input
- Propagating exception out of scope and back in
- Keeping a task alive

Interactive Data Input

```
with TEXT_IO; use TEXT_IO;
procedure Get_Input (Number : out integer) is

        subtype Input_Type is integer range  0..100;
        package Int_io is new INTEGER_IO (Input_Type);
        In_Number : Input_Type;

begin  -- Get_Input

        loop            -- to try again after incorrect input

                begin  -- inner block to hold exception handler

                        put ("Enter a number 0 to 100");
                        Int_io.GET (In_Number);
                        Number := In_Number;
                        exit;  -- to exit loop after correct input

                exception
                        when DATA_ERROR =>
                                put ("Try again, fat fingers!");
                                Skip_Line;    -- must clear buffer

                end;  -- inner block

        end  loop;

end  Get_Input;
```

Propagating Exception Out of Scope and Back In

```
declare
        package Container is
                procedure Has_Handler;
                procedure Raises_Exception;
        end Container;
         ----------
        procedure Not_in_Package is
        begin
                Container.Raises_Exception;
        exception
                when others => raise;
        end  Not_in_Package;
         ----------
        package  body  Container  is
                Crazy : exception;
                procedure  Has_Handler  is
                begin
                        Not_in_Package;
                exception
                        when Crazy => Tell_Everyone;
                end  Has_Handler;
                procedure   Raises_Exception  is
                begin
                        raise Crazy;
                end   Raises_Exception;
        end Container;
begin
        Container.Has_Handler;
end;
```

Keeping a Task Alive

```
task Monitor is
     entry Do_Something;
end Monitor;

task body Monitor is
begin
     loop      -- for never-ending repetition
          ...
          select
               accept Do_Something do

                    begin  -- block for exception handler
                         ...
                         raise Failure;
                         ...
                    exception
                         when Failure => Recover;
                    end;  -- block

               end Do_Something;  -- exception must be
                                   -- lowered before exiting
               ...
          end select;
          ...
     end  loop;

exception
     when others =>
          Termination_Message;
end Monitor;
```

Outline

- Overview
- Naming an exception
- Creating an exception handler
- Raising an exception
- Handling exceptions
- Turning off exception checking
- Tasking exceptions
- More examples

=> Summary

Summary

- Exception handling principles are consistent
- Suppression of exception checking will usually do more harm than good
- Use of exceptions must become a habit to be useful